DUBLIN PUBLIC LIBRARY
SCHOOL
30008

NAVAL POSTGRADUATE SCHOOL
Monterey, California



DISSERTATION

A3711

Construction of Optimal-Path Maps
for Homogeneous-Cost-Region Path-Planning Problems

by

Robert S. Alexander
Major, United States Army

September 1989

Dissertation Supervisor : Neil C. Rowe

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE

. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
DECLASSIFICATION / DOWNGRADING SCHEDULE	
PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
NAME OF PERFORMING ORGANIZATION Naval Postgraduate School	6b. OFFICE SYMBOL (If applicable) 37
7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School	
ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000
NAME OF FUNDING / SPONSORING ORGANIZATION USACDEC	8b. OFFICE SYMBOL (If applicable)
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER ATEC 88-86	
ADDRESS (City, State, and ZIP Code) Fort Ord, CA 93941	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

TITLE (Include Security Classification)
Construction of Optimal-Path Maps for Homogeneous-Cost-Region Path-Planning Problems

PERSONAL AUTHOR(S)
Alexander, Robert S.

TYPE OF REPORT	13b TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Ph.D. Dissertation	FROM _____ TO _____	1989 September	292

SUPPLEMENTARY NOTATION

COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Paths, Routes, Path-Planning, Shortest-Path Maps, Optimal-Path Maps, Weighted Region Problem, Wavefront Propagation

ABSTRACT (Continue on reverse if necessary and identify by block number)
Fast path-planning algorithms are needed for autonomous vehicles and tactical terrain-analysis tools. We explore a new approach using "optimal-path maps", that give the best path to a goal point from any given start point in cross-country two-dimensional terrain for a moving point of negligible size. Such maps allow fast point-location algorithms at run-time to categorize the start point according to the behavior of the optimal path to the goal, from which the path can be reconstructed. We study terrain modelled by piecewise-linear roads and rivers, polygonal obstacles, and by convex polygonal homogeneous-cost areas ("weighted regions"). We explore two methods for constructing optimal-path maps, one based on wavefront-propagation point-to-point path planning, and a more exact divide-and-conquer algorithm that reasons about how optimal paths must behave. In the exact approach, boundaries caused by terrain features are characterized using analytical geometry and optimal-path principles, and partial optimal-path maps are merged into complete ones.

DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL Prof. N. C. Rowe	22b TELEPHONE (Include Area Code) (408) 646-2462
	22c OFFICE SYMBOL 52Rp

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

☆ U.S. Government Printing Office 1986—606-24.

UNCLASSIFIED

T245459

Approved for public release; distribution is unlimited

Construction of Optimal-Path Maps for Homogeneous-Cost-Region Path-Planning Problems

by

Robert S. Alexander
Major, United States Army
B.S., United States Military Academy, 1974
M.S., Naval Postgraduate School, 1986

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

from the
NAVAL POSTGRADUATE SCHOOL
September 1989

ABSTRACT

Fast path-planning algorithms are needed for autonomous vehicles and tactical terrain-analysis tools. We explore a new approach using "optimal-path maps", that give the best path to a goal point from any given start point in cross-country two-dimensional terrain for a moving agent of negligible size. Such maps allow fast point-location algorithms at run-time to categorize the start point according to the behavior of the optimal path to the goal, from which the path can be reconstructed. We study terrain modelled by piecewise-linear roads and rivers, polygonal obstacles, and by convex polygonal homogeneous-cost areas ("weighted regions"). We explore two methods for constructing optimal-path maps, one based on wavefront-propagation point-to-point path planning, and a more exact divide-and-conquer algorithm that reasons about how optimal paths must behave. In the exact approach, boundaries caused by terrain features are characterized using analytical geometry and optimal-path principles, and partial optimal-path maps are merged into complete ones.

A3711
C.1

TABLE OF CONTENTS

I. INTRODUCTION....1

A. OVERVIEW OF PATH PLANNING....1

B. ASSUMPTIONS....2

C. THE OPTIMAL-PATH-MAP APPROACH TO PATH-PLANNING....5

D. SUMMARY OF RESEARCH REPORTED HEREIN....10

II. RELEVANT RESEARCH....11

A. APPLICABLE CONCEPTS FROM ARTIFICIAL INTELLIGENCE....11

B. APPLICABLE CONCEPTS FROM COMPUTATIONAL GEOMETRY....13

C. DEFINITIONS OF RELEVANT FREE-SPACE PATH-PLANNING PROBLEMS....25

D. TYPES OF PATH ERRORS....27

E. RELEVANT OPTIMAL-PATH PLANNING RESEARCH....28

III. MODIFYING WAVEFRONT PROPAGATION TO FIND SUB-OPTIMAL SOLUTIONS TO THE OPTIMAL-PATH-MAP PROBLEM....50

A. OVERVIEW....50

B. MODIFYING THE PATH-GENERALIZING FUNCTION FOR WAVEFRONT PROPAGATION OPTIMAL-PATH-MAP CONSTRUCTION....51

1. The Pure Version of Wavefront-Propagation Optimal-Path-Map Construction....51

2. The Diverging-Path Version of Wavefront-Propagation Optimal-Path-Map Construction....57

3. The Vertex-Edge Version of Wavefront-Propagation Optimal-Path-Map Construction....58

C. RECONSTRUCTING OPTIMAL PATHS FROM WAVEFRONT PROPAGATION OPTIMAL-PATH MAPS....65

IV. ANALYSIS OF WAVEFRONT-PROPAGATION OPM-GENERATION ALGORITHMS....66

A. SOURCES OF ERROR IN WAVEFRONT-PROPAGATION OPM-GENERATION ALGORITHMS....66

B. TIME COMPLEXITY OF WAVEFRONT-PROPAGATION OPM-GENERATION ALGORITHMS....67

C. SPACE COMPLEXITY OF WAVEFRONT-PROPAGATION OPM-GENERATION ALGORITHMS....68

D. EMPIRICAL PERFORMANCE OF WAVEFRONT-PROPAGATION OPM IMPLEMENTATIONS....69

V. CHARACTERIZATION OF REGION BOUNDARIES....72

A. REGION BOUNDARIES ASSOCIATED WITH PRIMITIVE TERRAIN FEATURES....72

B. A UNIFYING VIEW OF REGION BOUNDARIES....101

VI. ALGORITHMS FOR OPM CONSTRUCTION BASED ON SPATIAL REASONING....107

A. OPTIMAL-PATH TREE CONSTRUCTION....107

B. BASIC ALGORITHMS FOR ISOLATED TERRAIN FEATURES....107

C. EXTENDING THE BASIC ALGORITHMS TO MULTIPLE CONNECTED RIVER AND ROAD SEGMENTS....130

D. A DIVIDE-AND-CONQUER ALGORITHM FOR MULTIPLE-FEATURE OPM CONSTRUCTION....138

VII. ANALYSIS OF DIVIDE-AND-CONQUER EXACT-OPM ALGORITHM....143

A. SOURCES OF ERROR IN THE ALGORITHM....143

B. TIME AND SPACE COMPLEXITY OF THE ALGORITHM....144

C. EMPIRICAL PERFORMANCE OF THE ALGORITHM IMPLEMENTATION....149

VIII. CONCLUSIONS....150

A. GENERAL....150

B. COMPARISON OF WAVEFRONT-PROPAGATION TO SPATIAL-REASONING APPROACHES TO OPM CONSTRUCTION....150

C. USEFULNESS OF THE OPM APPROACH TO PATH PLANNING....151

D. AREAS FOR ADDITIONAL RESEARCH....152

APPENDIX A (THEOREMS)....153

APPENDIX B (BASIC WAVEFRONT PROPAGATION ALGORITHM)....205

APPENDIX C (WAVEFRONT-PROPAGATION OPM CONSTRUCTION SOURCE CODE)....208

APPENDIX D (HIGH-COST EXTERIOR-GOAL HCA INTERIOR-BOUNDARY CONSTRUCTION SOURCE CODE)....237

LIST OF REFERENCES....283

INITIAL DISTRIBUTION LIST....287

LIST OF TABLES

TABLE 1 - WAVEFRONT-PROPAGATION OPM ALGORITHM 52

TABLE 2 - WAVEFRONT-PROPAGATION OPM ALGORITHM CHANGES FOR VERTEX-EDGE VERSION 62

TABLE 3 - WAVEFRONT-PROPAGATION OPM-GENERATION, RELATIVE PERFORMANCE OF THREE VERSIONS 71

TABLE 4 - SUMMARY OF HOMOGENEOUS-BEHAVIOR-REGION BOUNDARIES BY TERRAIN TYPE. .73

TABLE 5 - BOUNDARY TYPES BY REGION ROOT PAIRS 106

TABLE 6 - OBSTACLE OPM ALGORITHM 109

TABLE 7 - RIVER-SEGMENT OPM ALGORITHM 112

TABLE 8 - ROAD-SEGMENT OPM ALGORITHM 113

TABLE 9 - HIGH-COST EXTERIOR-GOAL HCA OPM CONSTRUCTION ALGORITHM 115

TABLE 10 - HIGH-COST INTERIOR-GOAL HCA OPM CONSTRUCTION ALGORITHM 125

TABLE 11 - LOW-COST EXTERIOR-GOAL HCA OPM CONSTRUCTION ALGORITHM 128

TABLE 12 - LOW-COST INTERIOR-GOAL HCA OPM CONSTRUCTION ALGORITHM 130

TABLE 13 - MULTIPLE-CONNECTED-RIVER-SEGMENT OPM CONSTRUCTION ALGORITHM 136

TABLE 14 - MULTIPLE-FEATURE OPM CONSTRUCTION ALGORITHM 139

LIST OF FIGURES

Figure 1 - Example Optimal-Path Map 3

Figure 2 - Planar Partition with Corresponding Optimal-Path Tree 9

Figure 3 - Optimal-Path Map with Data Structures 14

Figure 4 - Trapezoid Method of Point Location 20

Figure 5 - Voronoi Diagram 24

Figure 6 - Taxonomy of Free-Space Path Planning Methods 29

Figure 7 - Wavefront Propagation 31

Figure 8 - Visibility Graph for RRR Algorithm 37

Figure 9 - Snell's Law for Optimal Paths 38

Figure 10 - Snell's Law Example 1 40

Figure 11 - Snell's Law Example 2 41

Figure 12 - Snell's Law Example 3 42

Figure 13 - Snell's Law Example 4 43

Figure 14 - Example of Pure OPM Version of Wavefront Propagation 55

Figure 15 - Example of Diverging-Path OPM Version of Wavefront Propagation 59

Figure 16 - Example of Vertex-Edge OPM Version of Wavefront Propagation 63

Figure 17 - Obstacle 75

Figure 18 - River Segment 77

Figure 19 - Characteristic Wedges 79

Figure 20 - Road Segment Example 1 81

Figure 21 - Road Segment Example 2 82

Figure 22 - HCA Opposite-Edge Sequences 86

Figure 23 - High-Cost, Exterior-Goal HCA Example 1 87

Figure 24 - High-Cost, Exterior-Goal HCA Example 2 88

Figure 25 - High-Cost, Exterior-Goal HCA Example 3. 89

Figure 26 - High-Cost, Interior-Goal HCA 94

Figure 27 - Low-Cost, Interior-Goal HCA Example 97

Figure 28 - Low-Cost, Exterior-Goal HCA Example 99

Figure 29 - Cost Function for Road Segment 102

Figure 30 - Construction of HCA Interior-Boundary Tree Example 1 119

Figure 31 - Construction of HCA Interior-Boundary Tree Example 2 120

Figure 32 - Construction of HCA Interior-Boundary Tree Example 3 122

Figure 33 - Multiple Connected River Segments 132

Figure 34 - Worst-Case Complexity of Multiple Connected River Segments 133

Figure 35 - Principle of Optimality for Path Planning 156

Figure 36 - Optimal-Path Turn Points 157

Figure 37 - Boundary Between Homogeneous-Behavior Regions with Point Roots 163

Figure 38 - Boundary Between Homogeneous-Behavior Regions with a Linearly-Traversed Root and a Point Root 165

Figure 39 - Boundary Between Homogeneous-Behavior Regions with Two Linearly-Traversed Roots 167

Figure 40 - Boundary Between Homogeneous-Behavior Regions Each with a Snell's-Law Edge as Root 169

Figure 41 - Boundary Between Homogeneous-Behavior Regions with a Snell's-Law Edge and a Linearly-Traversed Edge as Roots 174

Figure 42 - Boundary Between Homogeneous-Behavior Regions Each with Two Snell's-Law Edges as Roots 176

Figure 43 - Obstacle Shadow Boundaries 179

Figure 44 - Uniqueness of Obstacle Opposite Edge 181

Figure 45 - Obstacle Opposite-Edge Boundary 183

Figure 46 - River Segment Boundaries 185

Figure 47 - Road Segment Boundaries 189

Figure 48 - High-Cost Exterior-Goal HCA Boundaries 192

I. INTRODUCTION

A. OVERVIEW OF PATH PLANNING

Motion planning is an important problem in robotics, computational geometry, and many other applications. A central part of motion planning, known as *path* or *route planning*, is the process of determining the path to be taken either by an agent's appendages or by the entire agent. The research reported herein is concerned with the latter of these two path-planning processes. Specifically, it is concerned with planning paths over long distances in cross-country terrain. Thus we are not concerned with small-scale motion, where robot appendages are moved among objects on a work-bench or robot legs are placed on the ground, for example, nor with medium-scale motion, where the agent's path must be planned so as to provide adequate clearance for itself, but with large-scale motion, where the size of the agent is negligible compared with the surrounding terrain.

Path planning will not typically be the only, or even the most important task which competes for computing resources. For example, the purpose of an autonomous vehicle is to go somewhere independently and accomplish a mission, a task which will require a large number of intermediate tasks which will each take computing time and space. Therefore it is important to find path-planning algorithms which use as few resources as possible. This means increasing run-time speed and at the same time decreasing storage requirements. These are usually conflicting goals, but it is often possible to increase run-time performance or reduce storage at the expense of preprocessing time in a pre-mission phase when resources are not in demand.

The problem of finding an *optimal* (least-cost) *path* between two points for a negligibly small agent over fixed, two-dimensional terrain with known cost characteristics can be attacked by several methods. When the agent is constrained to travelling on a finite number of known paths, the problem can be solved by network search algorithms, a subject of thorough study in operations research. When the agent is not constrained to travelling on specified paths, the area is called *free space*. Path planning in two-dimensional free space is beginning to be studied in depth by researchers in such fields as artificial intelligence, robotics, and computational geometry. Most methods require homogeneous-cost background terrain interspersed with impassable obstacles, as for example for the Visibility-Graph algorithms [Ref. 1]. However, handling additional types of

terrain features (for example, linear low-cost features, e.g., roads, linear fixed-crossing-cost features, e.g., rivers, [Ref. 2] and polygonal regions of homogeneous-cost terrain, e.g., forests, swamps, or fields, [Ref. 3]) will improve the ability to model terrain realistically.

A promising approach to two-dimensional path-planning in free space which we develop in this research, called the *optimal-path-map* approach, provides greatly improved run-time speed at the expense of preprocessing time and storage. This approach partitions the plane during preprocessing into regions with similarly-behaved optimal paths and then locates a start point in this partition at run time. Figure 1 shows an example optimal-path map with boundaries separating the regions of similarly-behaved optimal paths. Additionally, a set of vectors is superimposed on the optimal-path map in Figure 1 showing initial directions of selected optimal paths. We develop the theoretical basis for such a partitioning for a more general set of terrain features than has previously been used in optimal-path-map construction, making this approach more practical for real-world cross-country path planning. Then once the optimal-path map is constructed, our approach can appeal to algorithms with worst-case time complexity of O(log n) to locate a start point in a planar partition (see Chapter II, Section B), where n is the number of terrain-feature vertices. Once the start point is located in the partition, the behavior of the optimal path is identified and the path can be reconstructed. This response time is very attractive, especially for real-time systems like missiles or for systems with many competing computing requirements like autonomous vehicles.

The principal results of our research are threefold. First, we adapted the wavefront propagation algorithm to find boundaries between regions of start points whose optimal paths are similarly behaved, and implemented three versions of the new algorithm. Second, we characterized boundaries mathematically by means of analytic geometry. Third, we proposed an algorithm to construct the planar partition using these mathematical results for convex polygonal and piecewise-linear terrain, as an alternative to our wavefront propagation algorithm.

B. ASSUMPTIONS

We assume that the terrain is known, and can be modelled by combinations of the five primitive types of terrain features presented below. We assume that terrain-feature edges can be modelled piecewise-linearly, that terrain is isotropic (traversal cost is independent of direction of travel), and that no two polygonal regions have common vertices. Although the mobile agent is constrained to travel in the two-dimensional plane of the

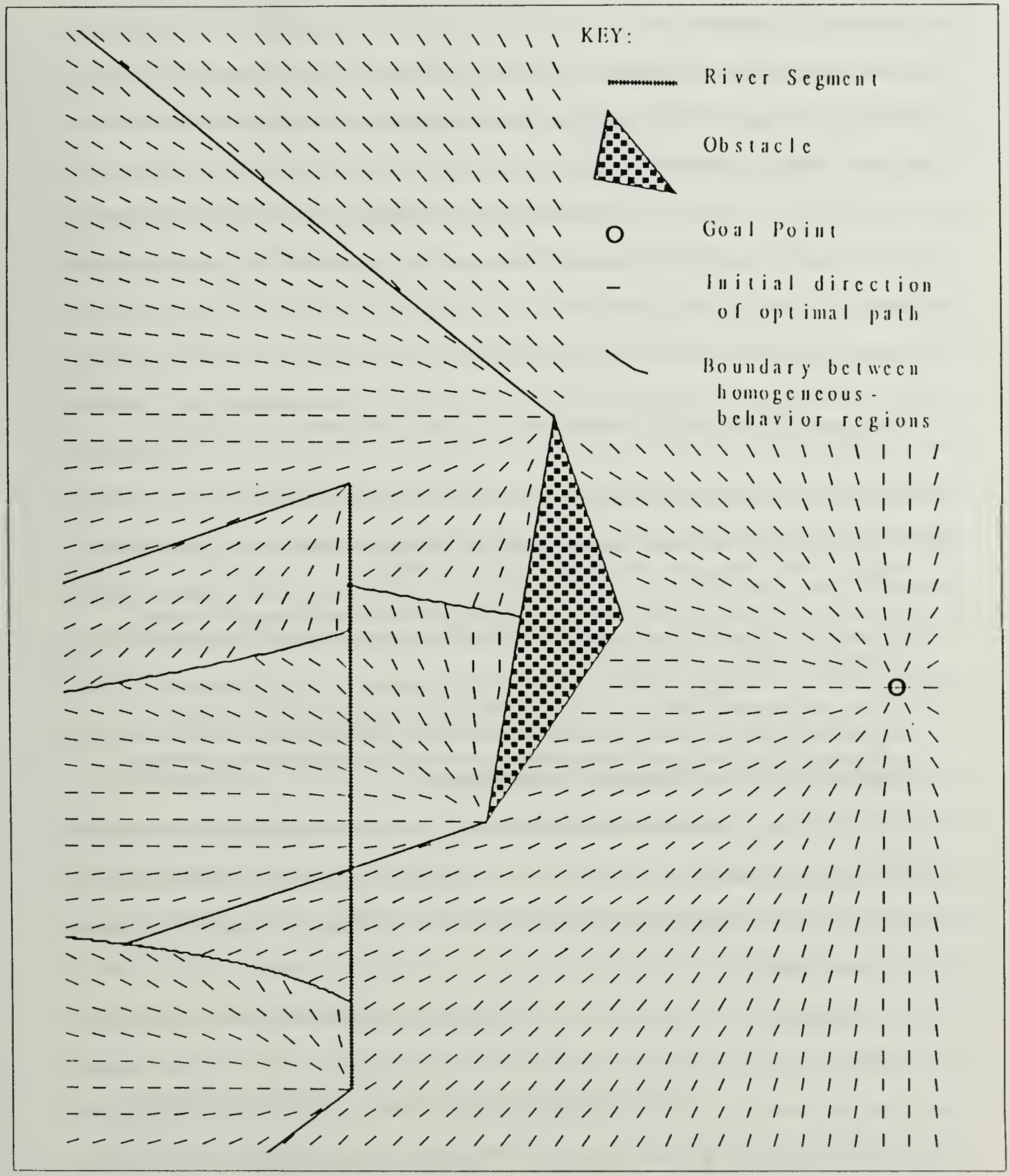


Figure 1

Example Optimal-Path Map

input map, assigned costs of travel may reflect that the actual surface being traversed varies in height. (See also Chapter II, Sections C and D.)

Following Mitchell [Ref. 4], we make the assumption called the *general-position assumption* (Appendix A, Assumption I-3), that no terrain-feature vertex lies on a homogeneous-behavior boundary generated by another terrain feature, i.e., that there is not an accidental alignment of boundaries with terrain-feature vertices. This restriction does not change the following results significantly, but allows the discussion to proceed without convoluted, but unimportant, conventions. In an actual implementation of the algorithms proposed below, this assumption must be retracted.

The following five terrain-feature types are allowed:

- **Background.** Areas of the map which do not contain other terrain features have a fixed cost per distance travelled.
- **Obstacles.** An *obstacle* is a convex polygon enclosing impassable terrain.
- **Rivers.** A *river segment* is a line segment whose cost to the agent to cross anywhere along its length is a fixed constant, not dependent on the angle of crossing.
- **Roads.** A *road segment* is a line segment with a fixed cost per distance for length-wise traversal. Thus a road segment is infinitesimally thin, can be crossed at no cost to the agent, and can be entered or left anywhere along its length.
- **Convex Homogeneous-Cost Areas.** A *convex homogeneous-cost area* (*HCA*) is a convex polygon with a constant positive cost per distance travelled. An HCA may have cost per distance greater or less than the background terrain, but not zero. The agent may enter or leave the area at any point on its circumference at no additional cost.

These terrain-feature types could all be modelled by HCAs. However, allowing obstacle, river, and road terrain features enhances efficiency by allowing us to take advantage of their simplicity. Specifically, it is an advantage to avoid, where possible, the complicated analysis of paths through homogeneous-cost regions (see Chapter II, Section E2b(3)).

How realistic are the above assumptions? There are at least three issues. First, is it reasonable to expect that we know the characteristics of the terrain; second, can terrain be adequately modelled by piecewise linear curves; and third, will the use of convex non-adjacent polygons be sufficient? As discussed in Chapter II, Section D4, the Defense Mapping Agency and other U.S. Government agencies currently have the ability to produce maps which characterize terrain according to the speed at which a given vehicle type can traverse it. (Of course, cost in terms of time is the reciprocal of speed.) The program used to produce these maps, called

Army Mobility Model (AMM), takes as input a digitized combination of soil conditions, vegetation, man-made features, and elevation which is available at present only for selected areas of the earth, but there is an ongoing effort to expand this database. As this database is expanded, AMM will be able to produce cost maps of more of the world's surface, so that a path-planning system which uses AMM cost maps as input can be expected to know the characteristics of the terrain. However, an additional consideration is that terrain may be impermanent. In this case our assumptions will be invalidated.

The second issue is whether terrain can be adequately modelled using only piecewise linear curves. Computational geometry relies very heavily on the use of piecewise linear curves to approximate reality, since there is a fixed precision associated with any computer, and a finite amount of storage. In fact, the very concept of continuity is a mathematical abstraction, since at some level the most smoothly continuous curves will be seen to degenerate into discrete elements. For example, a wood-line may seem to form a continuous curve, when in fact at the scale of individual trees it is clearly discontinuous. Since the database maintained by the Defense Mapping Agency has a maximum resolution of 12.5 meters square, we can be assured that no representation we propose will be more accurate than this. One additional consideration is that small nuances of the terrain will normally have much less effect on optimal paths than will large features. Of course, it is always desirable from the viewpoint of efficiency to use as few line segments as possible to approximate a curve in order to reduce the number of terrain-feature vertices in the input map.

The third question is much more of a problem. The use of convex polygons will clearly not approximate all types of terrain if we require that no two polygons have common vertices. The output of Army Mobility Model for example, allows non-convex polygons. This research uses the non-adjacent-convex-polygon assumption in order to attack a problem of somewhat smaller scope first, with the intention of expanding the scope in the future to incorporate non-convex regions. The next step will be to extend the analysis of Chapter V to include the case of adjacent convex polygons.

C. THE OPTIMAL-PATH-MAP APPROACH TO PATH-PLANNING

The optimal-path-map approach to path planning groups paths according to their general behavior with respect to a goal point. A surjective function is defined to map optimal paths to generalizing path descriptions so that paths with similar behavior are mapped to the same description. The usual definition of "similar be-

havior" is crossing the same sequence of terrain-feature vertices and edges. Boundaries are constructed to partition the plane of the map into regions whose start points have similar behavior. Then to determine an optimal path, a given start point is located within the partition. The path description of the region associated with the start point applies to the optimal path from the start point, so this path description is specialized for the given start point to give an optimal path. The focus of our research is the construction of the planar partition.

How can paths be represented so they can be grouped according to their behavior? Theorem I-2 states that optimal paths among piecewise-linear and polygonal terrain are always piecewise linear, changing direction only at terrain-feature vertices and edges. This fact suggests two possible ways to represent optimal paths. The more natural way to represent a single piecewise-linear path would be by listing the coordinates of its turn points. Alternately, we could list the terrain-feature vertices and edges at which a path turns. The first representation has the difficulty that there is no immediate way to tell from the list whether or not turn points from two different paths lie on the same terrain-feature edge. The second representation allows paths to be grouped more easily according to whether they cross the same terrain-feature edges and vertices, but has the difficulty that it is not clear by looking at the list what the coordinates of a turn point are on a terrain-feature edge. This conflict suggests a composite representation wherein a list contains terrain-feature vertices and edges, and for each edge, may also contains as supplemental information the exact coordinates at which the path crosses the edge. This is the representation we adopt, calling such a list a *path list.*

The path list can be used to represent a specific optimal path as well as a generalized description of an optimal path. If a path list has a terrain-feature vertex as its first element, the path is completely determined because it will go from the start point directly to the vertex, from where a unique path goes to the goal (Corollary I-3.1, Appendix A). If a path list has an edge as its first element and no supplemental information is included with that edge, the path list represents all optimal paths whose first turn point lies on that edge. If however, coordinates of the crossing point are included with the edge, the path is completely determined. When listing an edge in a path list, it is also important to distinguish between edges crossed from different directions, because for example, paths may enter the same portion of a road from both sides; we want to distinguish between the two sets of paths which come from either side of the road. For consistency in our discussions, we adopt the convention that for a start point with no feasible paths (for example, a start point inside an impassable obstacle), the optimal-path list is a null symbol concatenated with the goal point.

Now the path-generalizing function can be defined more fully for the usual definition of similar behavior of paths. For the set O of all optimal paths and the set $(V \cup E)^*$ of all combinations of terrain-feature vertices and edges, the function $f : O \rightarrow (V \cup E)^*$ maps an optimal path to its path list.

Define a *homogeneous-behavior region* with respect to a goal G as the set of all start points whose optimal paths are mapped by the path-generalizing function to the same set. Thus, start points whose optimal paths have the same path lists are considered to be in the same homogeneous-behavior region for the usual definition of the path-generalizing function. Define the *root* of a homogeneous-behavior region as the first element of the path list associated with the region. Since a root may represent a terrain-feature edge which can be crossed at any point along its length, the supplemental information cannot be retained by the path list associated with the root. Define a *homogeneous-behavior boundary* as the locus of points lying in two homogeneous-behavior regions. On a homogeneous-behavior boundary (except for obstacle edges), at least two optimal paths exist for a given point.

The fundamental principle upon which spatial reasoning about optimal paths is based is the principle of optimality. In its general sense, the principle of optimality states that if it applies to a system, future optimal policy in the system depends only on its current state and not on its past history. Theorem I-1 (Appendix A) states that the principle of optimality applies to the path-planning domain. In other words, it states that the portion of an optimal path from any point on the path to the goal is also an optimal path.

We extend the general-position assumption discussed above to terrain feature edges by adopting the convention that any terrain feature edge intersected by a homogeneous-behavior boundary is to be treated as two distinct edges, one on each side of the boundary. The immediate result of this assumption, the principle of optimality, and Theorem I-2, is the uniqueness of optimal paths from any terrain feature vertex or across the interior of any edge. (Corollary I-3.1, Appendix A.) It follows from the definitions of homogeneous-behavior regions, roots, and boundaries, the general-position assumption, and Theorem I-2 that there is a unique root associated with each homogeneous-behavior region (Corollary I-3.2). It also follows that homogeneous-behavior regions are "star-shaped" with respect to the region root (Corollary I-3.3).

An *optimal-path tree* of a set of terrain features with respect to a goal point is the index tree for all possible path lists. In other words, it is the tree whose root represents the goal and whose internal nodes are terrain-feature vertices and edges, such that for each node, the optimal paths from that node's vertex or edge go

first to the vertex or edge represented by the node's parent. Therefore, the path list for the vertex or edge associated with a node is found by following the parent pointer of the node back to the root of the tree, which is the goal. Each node of the tree corresponds to a unique homogeneous-behavior-region root, which corresponds to a unique homogeneous-behavior region. Thus, locating a start point in a region of the planar partition is equivalent to specifying which node of the tree identifies the behavior of the optimal path from that start point. Figure 2 shows an example planar partition with its corresponding optimal-path tree.

An initial version of the optimal-path tree can be constructed by using a point-to-point path planner to compute the optimal path from each terrain-feature vertex on the input map and then inserting the turn points of each resulting optimal path into a tree. The method presented in Chapters V and VI uses the optimal-path tree to construct the planar partition, and revises it by inserting nodes which correspond to terrain-feature edges. However, the method presented in Chapter III constructs the optimal-path tree at the same time as it constructs the planar partition.

An *optimal path map* or *OPM* is a partition of the plane into homogeneous-behavior regions with respect to a goal, along with its associated optimal-path tree. There is a finite optimal-path tree associated with every two-dimensional map consisting of terrain as defined above (Theorem I-4, Appendix A). The specification of this optimal-path tree is a necessary part of the optimal-path map, and we will assume that the term optimal-path map implies both the representation of the planar partition and of the optimal-path tree, with some means of linking each node with its corresponding homogeneous-behavior region in the partition. A typical representation of the planar partition is the doubly-connected-edge-list discussed in Chapter II, Section B.

Several partitioning algorithms for terrain containing only obstacles (the binary case) have been proposed in an attempt to present faster solutions to the point-to-point path-planning problem (see Chapter 2), and several algorithms even solve a portion of the optimal-path-map problem with respect to weighted regions by creating the optimal-path tree in pursuit of single-path solutions. In this research, we investigate the problem of creating an optimal-path map for weighted-region terrain, focusing on a solution to the optimal-path-map problem as an end in itself. We choose to investigate this approach because it offers the most opportunity for enhancement of run-time performance at the expense of preprocessing time because of the promise of O(log n) run-time complexity to identify an optimal path for a map of n terrain-feature vertices.

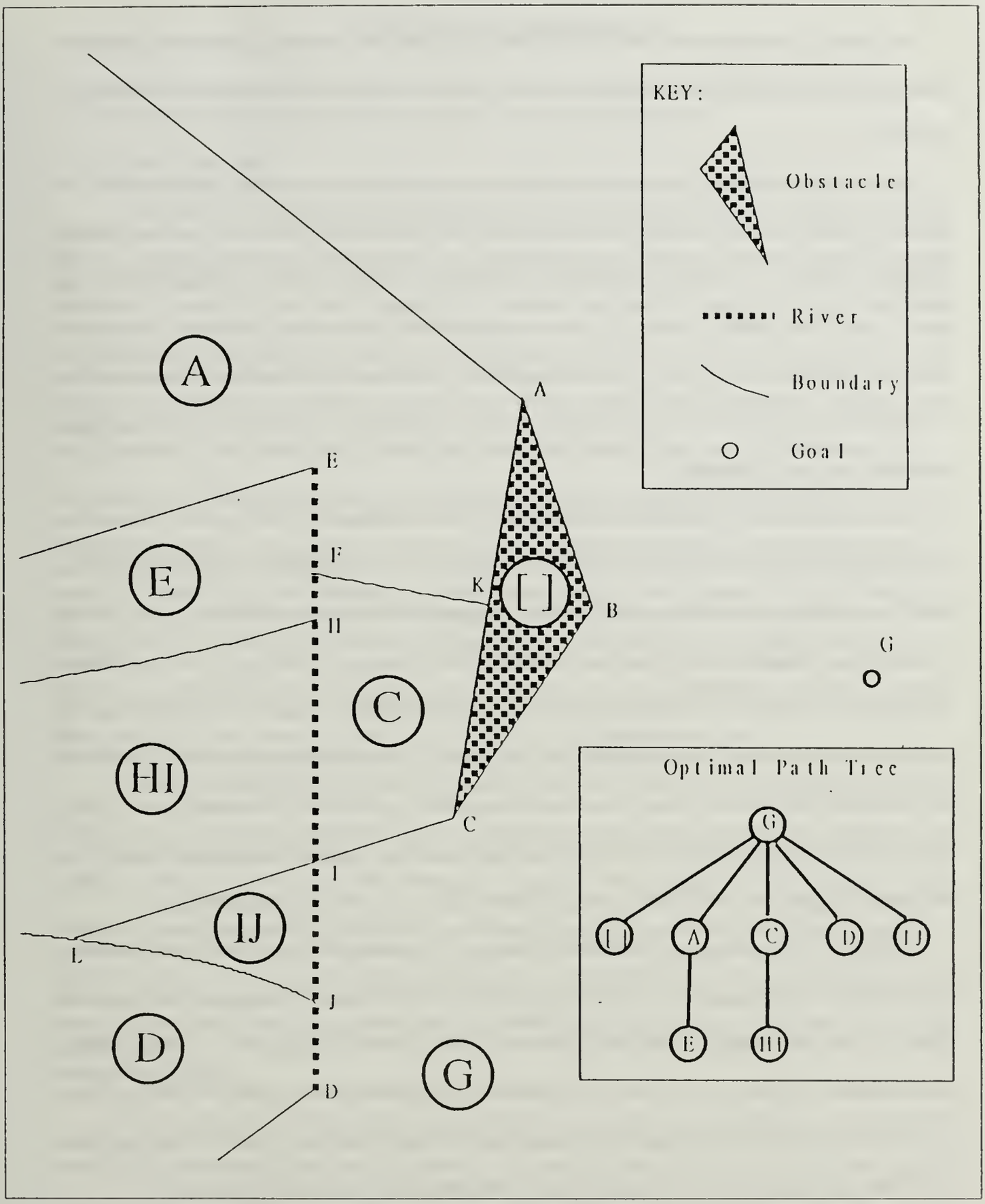


Figure 2

Planar Partition with Corresponding Optimal-Path Tree

D. SUMMARY OF RESEARCH REPORTED HEREIN

In Chapter II, we explain and classify the path-planning algorithms relevant to this research, specifically those dealing with negligible-sized agents in a fixed, known environment where terrain is two-dimensional, free space.

One method of obtaining sub-optimal solutions to the optimal-path-map problem uses a version of the standard wavefront propagation algorithm. Such an algorithm is presented in Chapter III, and the extensions necessary to create optimal-path maps are developed. Chapter IV is an analysis of the algorithm in the previous chapter. Two primary sources of error are examined, and known results of inherent inaccuracy in wavefront propagation are extended to the resulting OPMs. The theoretical time and space complexity of the above algorithm is presented, along with empirical results concerning execution times for three alternative heuristics used with the algorithm.

A second approach to solving the two-dimensional optimal-path-map problem is to reason about how optimal paths behave in the presence of various terrain features. This reasoning leads to analytical characterization of the boundaries between homogeneous-behavior regions of similarly-behaved optimal paths as functions of terrain feature characteristics. It turns out that all boundaries generated by the roads, rivers, and obstacles are segments of conic sections. Other boundaries are more mathematically complex, and in many cases cannot be described in closed-form expressions. First in Chapter V, a set of definitions is presented, followed by development of the characterizations of boundaries generated by "primitive" terrain feature types, i.e., single polygons and single line-segments. Then the characterization of more complex combinations of primitive terrain features is discussed, and decomposability is defined for construction of optimal-path maps.

In Chapter VI, algorithms use the results of Chapter V to generate OPMs more accurate than those of Chapter III for isolated occurrences of each type of primitive terrain feature. Then an algorithm based on the divide-and-conquer paradigm is presented to generate OPMs for some "decomposable" maps with multiple terrain features. In Chapter VII the divide-and-conquer exact-OPM algorithm is analyzed, first in terms of sources of error, and then with respect to theoretical time and space complexity. Then the empirical performance of an implementation is discussed. Chapter VIII summarizes the results of the research.

II. RELEVANT RESEARCH

A. APPLICABLE CONCEPTS FROM ARTIFICIAL INTELLIGENCE

1. Search Methods

One of the central problem-solving techniques in Artificial Intelligence is the use of search [Ref. 5], [Ref. 6]. A search problem is couched in terms of a current state and a goal state, operators are defined which transform the system from one state to another, and a search is conducted for a sequence of operators which will transform the current state to a goal state. Conceptually, a *search space* is a directed graph whose nodes represent all possible states, and whose edges represent operators. Solving the problem means applying graph-search algorithms in the search space to find a path from a start node to a goal node. The search space may be a very large, even an infinite graph which is not represented explicitly, but as the algorithm proceeds, it creates a sub-graph, called a *search graph* (or *search tree*), whose nodes are the states actually reached during the search. The underlying aim is to find ways to make the search graph as small as possible while still including the goal state, i.e., to look at as few states of the search space as possible on the way to finding the goal. There are two ways of limiting the size of the search graph. One way is to guide the search by means of heuristics, and the other is to represent the problem in such a way as to reduce the search space.

When no domain-specific information is used to guide decisions about which node of the search graph to process next, the process is called *blind search*. Although few problems have a search space small enough to allow practical use of blind search, the techniques used provide the foundation for *heuristic search*, where information is used to guide the search. All the search techniques discussed below can be said to conform to a general model where the search is initialized by placing an initial node on an agenda, and proceeds by expanding the first node on the agenda, putting the node's children on the agenda in a manner which varies from technique to technique.

Branch-and-bound search, also known as *Dijkstra's algorithm*, is a generalization of breadth-first search which uses heuristic information. The distance of a node from the start is not measured by the number of edges from the node to the root as in breadth-first search, but by the total cost of the edges. Thus, each edge has an associated cost, and at each iteration, after a node has been expanded and its children placed in the agen-

da, the agenda is sorted by cost to keep lower-cost nodes first. Since physical distance is the normal metric in the path-planning domain, this is a natural search technique to use. This technique guarantees that the first path found to the goal is the lowest-cost solution.

Another search strategy which is widely used in path planning is called A* search. It sorts its agenda according to the sum of the cost function and evaluation function at each node. If the evaluation function value from any point to the goal is a lower bound on the actual cost from the point to the goal, it is guaranteed that the first time the optimal path to the goal is selected from the agenda it will be recognized as optimal.

2. Domain-Specific Heuristics as Guides to Search

General solutions to problems tackled by Artificial Intelligence researchers are usually so difficult that great advantages are to be gained by finding rules-of-thumb to focus the search in the right direction. Heuristic search strategies use cost and/or evaluation functions to guide the search. Rich [Ref. 7] states that the field of artificial intelligence is largely the study of heuristic search for solving difficult problems, and The Handbook of Artificial Intelligence calls heuristic search "one of the key contributions of Artificial Intelligence to efficient problem-solving" [Ref. 5]. In the path-planning domain, there is a natural heuristic which is often used to guide search for an optimal-cost path, which is that for a path from the start point to an intermediate point, if the intermediate point is closer in straight-line distance to the goal than some other intermediate point from another path (irrespective of terrain yet to be negotiated),the first path is preferred over the second for further exploration.

3. Problem Representation

It is often the case in problems studied in artificial intelligence research that a problem which seems very difficult when represented in one way will suggest a natural solution when represented in a different way. In other words, finding a good problem representation is often the key to efficient solution of the problem, as well as to clear understanding of the problem on the part of researchers [Ref. 5]. Path-planning algorithms, for example, are essentially ways of transforming an infinite search space to a finite one, and then searching the transformed search space using one of several heuristic-aided search algorithms discussed above.

B. APPLICABLE CONCEPTS FROM COMPUTATIONAL GEOMETRY

1. Definitions for Optimal-Path Maps

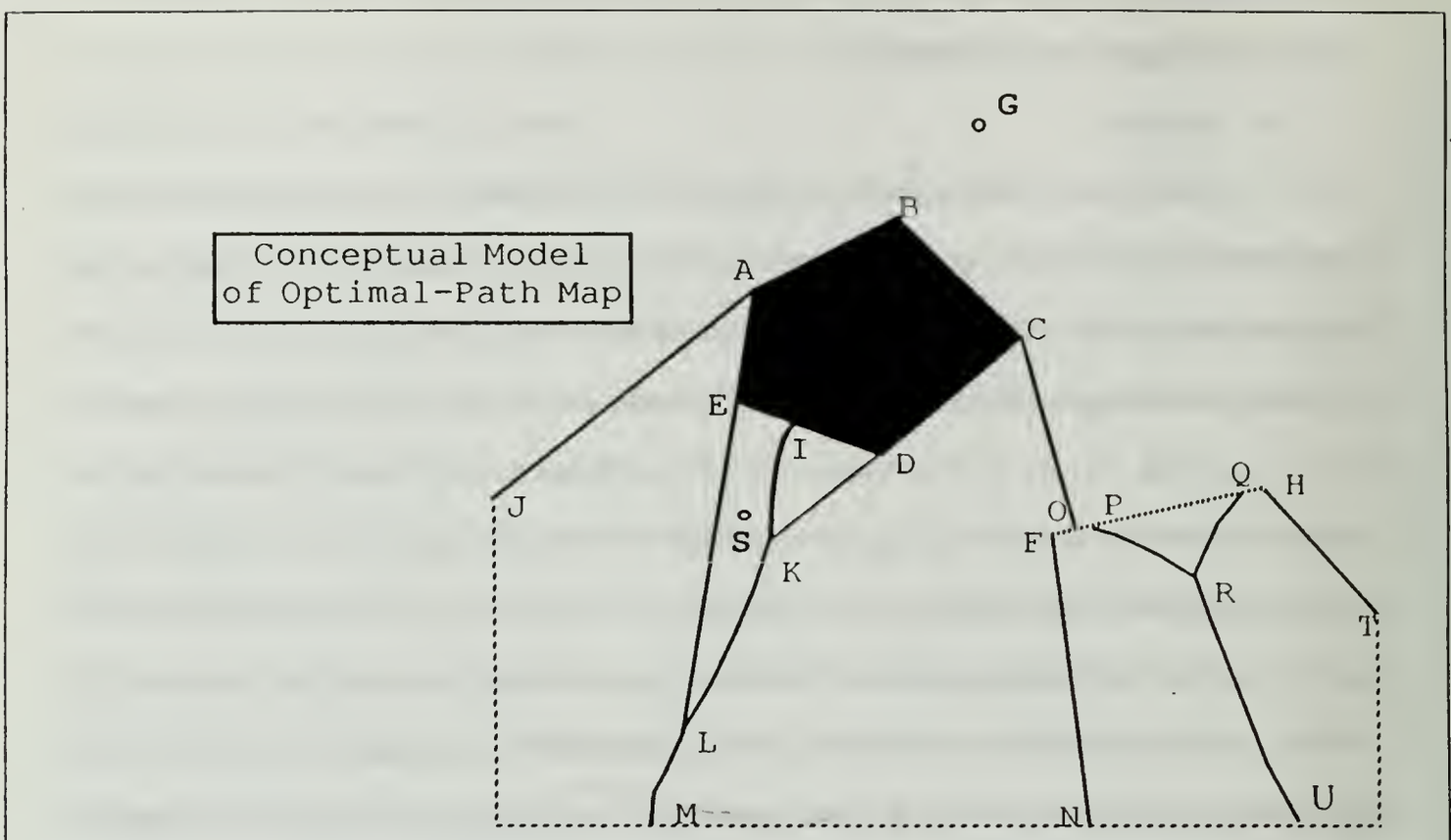
a. Path List

When optimal paths are guaranteed to consist of line segments between a finite number of turn points, which Theorem I-2 shows is true of the terrain considered in this research, they can be represented by listing these turn points. It is also shown in Theorem I-2 that these turn points occur only at terrain-feature vertices and edges. This suggests two possible ways to list the turn points. The most direct way is to list the coordinates of the points. This allows direct reconstruction of the path from its list. However, this representation makes it somewhat more difficult to compare two lists to determine if the paths they represent cross the same edges. It might be better to list explicitly the vertices and edges that a path crosses. This representation has the drawback, however, that some computation would be necessary to determine for each edge crossing exactly where the crossing occurred. Since our research is primarily concerned with grouping paths together according to their general behavior, we adopt the second representation, calling such a list a *path list.* An example path list from start point S to goal point G in Figure 3 is [E,A,G], while from point R there are three possible good path lists of [F,C,G], [H,G], and [PQ,G]. For consistency in later discussions, we say that for a start point with no feasible paths (for example a start point in the center of an impassable obstacle), the path list consists of a special null symbol concatenated with the goal point.

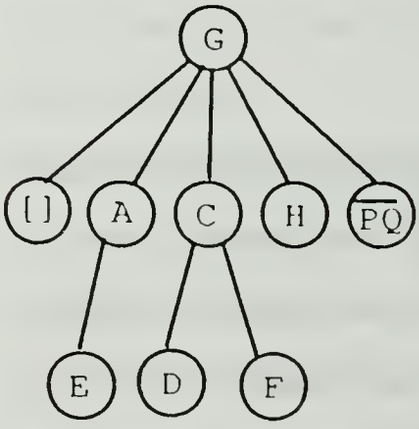
b. Path-Generalizing Function

The concept upon which the optimal-path-map approach to path planning is based is that paths can be grouped according to their behavior. A *path-generalizing function* $f:O \rightarrow B$ is defined from the set of optimal paths to the set of behaviors of optimal paths, which maps an optimal path to a description of its behavior. Since many paths may share the same behavior descriptions, f is a surjective function. The usual way to define the behavior of a path is by listing the vertices and edges it crosses. In that case $B = (V \cup E)^*$, the set of all combinations of terrain-feature vertices and edges. Since path lists are defined in terms of vertices and edges, the usual definition of f is that it maps an optimal path to its path list.

A path-generalizing relation R which relates two points if the path-generalizing function maps their optimal paths to identical path lists is an equivalence relation because the identity relation is in general



Optimal-Path Tree



Planar Partition (DCEL)

edge	V1	V2	R1	R2	P1	P2	algebr form	addntl parameters
AB	A	B	G	[]	AJ	BC	line	-
BC	B	C	G	[]	AB	CD	line	-
⋮								
AJ	A	J	G	A	AE	AB	line	-
⋮								
IK	I	K	D	E	ID	KL	hyperl	a,b
⋮								
JM	J	M	A	G	AJ	LM	-	-
⋮								

Figure 3

Optimal-Path Map With Data Structures

an equivalence relation. Since the domain of f is the set of all points on the input map, f induces a partition on the plane of the input map through this equivalence relation.

c. Homogeneous-Behavior Region

Define a *homogeneous-behavior region* with respect to a goal point and a path-generalizing function as the set of start points whose optimal paths to that goal point are mapped by the path-generalizing function to the same path behavior. In our work, this is equivalent to saying that it is the set of start points whose optimal paths have the same path lists. Each homogeneous-behavior region corresponds to an equivalence class of the path-generalizing relation R, and so is a subdivision of the partition induced by R on the plane of the input map. In Figure 3, for example, point S is in the homogeneous-behavior region enclosed by segments EI, IK, KL, and LE.

An *optimal-path map* (OPM) is defined as the partition of the plane of the input map into homogeneous-behavior regions, along with their associated path lists. For the conceptual representation of an optimal-path map shown in the top half of Figure 3, the two data structures in the bottom half of the figure fully specify the OPM.

d. Homogeneous-Behavior-Region Root

Because of the definition of homogeneous-behavior regions, each unique path list defines a homogeneous-behavior region. Thus given a path list, the associated region is defined. By the general-position assumption (Assumption I-3, Appendix A), there will be no accidental alignment of boundaries from another region such that there is more than one path list from a region. The first element of the path list associated with a region is defined as the *homogeneous-behavior-region root*. For example, in Figure 3 the path list of start point S with respect to goal point G is [E,A,G], and point E is the region root of the region of which S is a member.

2. Data Structures

Several data structures with wide utility in computational geometry are useful in the optimal-path-map domain. Since an optimal-path map consists of the set of path lists from homogeneous-behavior-region roots and a planar partition, these two items must be represented.

a. The Optimal-Path Tree

The optimal-path tree is a way of representing a set of path lists. It is a direct extension of the *shortest-path tree* concept [Ref. 8]. An optimal-path tree (OPT) is a tree whose root (not the same as a homogeneous-behavior-region root) is the specified goal point, whose nodes are terrain-feature vertices and edges, and for which an optimal path from the terrain-feature vertex or edge represented by any node in the tree goes directly to the vertex or edge represented by that node's parent. Each node of the tree corresponds to a homogeneous-behavior region, and every homogeneous-behavior region is represented by a node. (see Theorem I-4, Appendix A). Thus by labeling regions and OPT nodes the same, or by establishing pointers from regions to nodes of the OPT, a linkage is established which allows retrieval of the appropriate OPT node given a region. Then the path list associated with the region can be reconstructed by tracing upwards through the tree to the tree's root. Note, however, that further computation usually is necessary to reconstruct the optimal path from the path list by finding optimal edge crossings. Another important characteristic of the optimal-path tree is that it reduces the redundancy of storage of optimal paths associated with terrain-feature vertices and edges by integrating them all into one structure. In Figure 3, the optimal-path tree is shown for the given terrain map.

b. The Doubly-Connected Edge List (DCEL)

A planar partition could be represented in edge-list form in which, for each vertex of a piecewise-linear approximation of the boundary between subdivisions of the partition an ordered list of its incident edges is given. Although this is a natural representation, some of the information implicitly present could be explicitly listed, enhancing efficiency at the expense of preprocessing time and storage. A doubly-connected edge list is such a data structure that has proven to be quite useful in representing a planar partition. Represent each edge as a node in the DCEL, and label each edge, vertex, and region. Note that the terms edge and vertex as used in connection with the DCEL refer to piecewise-linear homogeneous-behavior-region boundary edges and vertices, not to terrain-feature edges and vertices. With each edge-node, associate a six-tuple of data elements $(V_1,V_2,R_1,R_2,P_1,P_2)$. The V_i are the two vertices of the edge. The assignment of vertices to the two fields V_1 and V_2 is arbitrary, but once assigned is fixed. Once the vertices are assigned, the edge becomes directed from V_1 to V_2, R_1 is the region (or face in the terminology of computational geometry) to the left of the edge, and R_2 is the region to the right. P_1 is a pointer to the edge-node which is adjacent to edge V_1V_2 in

a counterclockwise rotation about V_1, and similarly for P_2 with respect to V_2. A partial listing of the DCEL for the optimal-path map of Figure 3 is also shown. [Ref. 8]

For a DCEL representing a partition with n vertices, a single pass in time O(n) can create arrays of headers of vertex and region linked lists, so that straightforward algorithms can retrieve the sequence of edges incident on a vertex or enclosing a region, in time proportional to the number of edges involved. A graph in edge-list form can be transformed to a DCEL in time O(n). [Ref. 8]

An extension of the DCEL allows curved edges, as well as piecewise-linear ones, to be represented. Additional fields for each edge-node can be added to the DCEL to represent the algebraic form of the curve and to represent additional parameters necessary to specify the curve analytically. For example, if a curve represented a segment of a hyperbola, the entry in the first additional field would note that, and the second additional field would contain the two parameters of the equation of a hyperbola. Two points on the hyperbola, the endpoints of the segment, are listed in the DCEL, so the hyperbola segment is fully specified.

3. The Plane Sweep Paradigm

Many algorithms in computational geometry follow the *plane sweep* paradigm. The idea is to process a geometrical structure in the plane in an ordered fashion, normally by sweeping an imaginary vertical line from left to right from event point to event point, where an event point is a point in the plane at which some action may need to be taken. Two data structures are useful in conducting a plane sweep, an *event-point schedule* and a *sweep-line status.* At any point along the sweep axis, the geometrical structure is characterized by a status which is the relation of the vertical line to the geometrical structure. For example, the status may be an ordered list of line segments of the structure which intersect the sweep line. This status will change at a finite number of points along the sweep axis for a finitely-describable structure. These changes in status are the places at which the problem must be processed or analyzed. These points along the sweep axis are maintained in the event-point schedule. The event-point schedule is often some form of a queue. [Ref. 8]

4. Point-Location in the Cartesian Plane

Linked to any algorithm that partitions the Cartesian plane in order to represent properties of points in each region is the requirement to retrieve those properties when queried about any point in the plane specified by its coordinates. Algorithms that build optimal-path maps are partitioning the plane into regions such that each region contains those start points with similarly-behaved optimal paths to a given goal-point. It is neces-

sary to determine in which region the point lies. If the boundaries between regions are piecewise-linear curves, there are several algorithms from computational geometry which can be used to locate a point in the planar partition.

The *slab method* of point location in a planar partition draws a horizontal line through each vertex of the partition, and then sorts the regions (or *slabs*) lying between horizontal lines from top to bottom during preprocessing. This allows location of the point within a slab in O(log n) time by use of bisection search based on the y-coordinate of the point, where n is the number of vertices in the partition. Line segments which comprise the boundaries of the partition cross through each slab. Within a slab they can be ordered from left to right because at no point in the interior of a slab do two line segments intersect, since the slabs were defined by drawing horizontal lines through all the intersection points of the partition. Then bisection search can be used to locate the point horizontally between line segments within the slab in O(log n) time, for a total location time of O(log n + log n) = O(log n). Two disadvantages to this method are the requirement for preprocessing time and storage space. Preparata and Shamos show how to reduce the basic $O(n^2 \log n)$ preprocessing time to $O(n^2)$ using a plane sweep approach, but the algorithm requires at worst $O(n^2)$ space. [Ref. 8]

A second point-location method is the *chain method.* Instead of dividing the planar graph horizontally with slabs, it finds vertical *chains*, or connected line segments, of edges, which are monotone with respect to the y-axis, i.e., such that no two points on the chain have the same y-coordinate. It then constructs two binary search trees, the first having those chains as nodes and the second having segments of chains as nodes. The two trees can be traversed in $O(\log^2 n)$ time to locate a point. A DCEL can be preprocessed in O(n log n) time into the two binary search trees. which take at worst O(n) space. [Ref. 8]

Another point-location method is the *triangulation refinement method.* A set of connected line segments is said to be triangulated if each vertex is connected by a line segment with at least two other vertices, i.e., the line segments all form triangles. The planar partition is triangulated in O(n log n) time by standard methods from computational geometry, and a hierarchy of triangulations is constructed upon which to search. This method leads to O(log n) query time, O(n log n) preprocessing time, and O(n) storage. [Ref. 8]

An extension of the chain method, the *bridged chain method*, uses an elegant method that permits search in O(constant) time for subsequent searches, after a higher cost for a first search. It happens that the chain method meets the conditions for application of the bridging technique, and so bridging is used to ac-

cumulate information during the search process. This technique brings the chain method to efficiency comparable with the triangulation refinement method. [Ref. 8]

Although the above two methods achieve the theoretically optimum worst-case bounds, there may be sub-optimal methods which afford better practical performance. Specifically, the *trapezoid method*, which could be considered an extension of the slab method, gives an O(log n) query which always succeeds in fewer than $4\lceil \log n \rceil + 3$ tests, and uses O(n log n) storage and preprocessing time. Actually, average-case storage may be O(n). This method has the additional property that it may be extended to curvilinear edges, so it may be especially useful in our application since instead of approximating curves piecewise-linearly, they may be represented exactly by their analytical form. [Ref. 8]

A problem with the slab method was the $O(n^2)$ worst-case space complexity, where n is the number of vertices of the graph representing the planar partition. This problem was due to the possibility that edges could span most of the horizontal slabs, each such edge needing to be segmented into O(n) fragments. In the trapezoid method, it can be shown that no more than 2 log n fragments will ever be needed for any edge, so no more than O(n log n) space is required. The trapezoid method defines a trapezoid as having two horizontal sides and two other sides which may be unbounded, or else if they exist are edges of the graph not interrupted by vertices. The basic operation of the algorithm is to split a trapezoid into subordinate trapezoids. The progress of the splitting algorithm is paralleled by the building of a balanced binary search tree which represents a hierarchy of subordinate trapezoids. This tree can then be searched to locate a point in a trapezoid. Figure 4 (adapted from Preparata and Shamos [Ref. 8]) shows an example trapezoid with its corresponding search tree.

The splitting operation for the trapezoid method proceeds by finding the median y-coordinate among the vertices contained in the current trapezoid T and dividing T into two "slices" T_1 and T_2 by drawing a horizontal line through the median vertex. Then those edges which intersect the top or bottom horizontal side of T are scanned from left to right, and the first edge which also intersects the newly drawn horizontal line, i.e., which spans T_1 or T_2, defines a new trapezoid T_3. The scan continues until all edges which span T_1 or T_2 are found, with a new trapezoid being generated for each spanning edge. Note that edge e_1 defines the first new trapezoid T_3 in Figure 4 because it spans the top and median lines of T. T_3 will not need to be further split because there are no vertices contained in it. Spanning-edge e_2 is found next, and creates T_6. Finally e_3 is

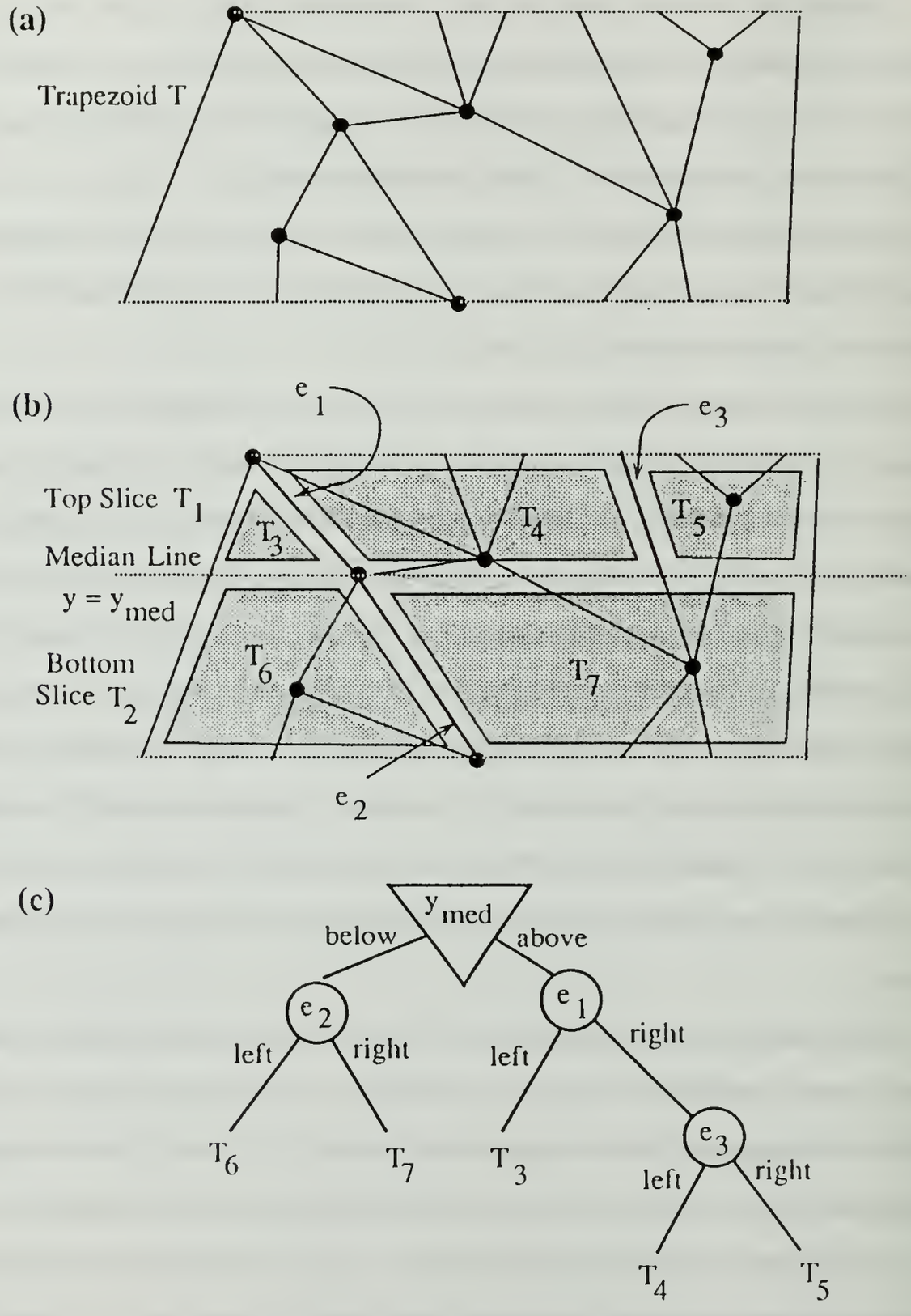


Figure 4

Trapezoid Method of Point Location

found, creating T_4. No other spanning edges are found, so T_5 and T_7 are also defined. T_4, T_5, T_6, and T_7 all contain vertices of the graph in their interiors, so they will need to be split further in subsequent iterations of the algorithm. Triangular nodes of the search tree represent horizontal splits at graph vertices, while circular nodes represent the definition of new trapezoids by spanning edges. There will be n-2 triangular nodes of the tree, one for each except the left and right extreme vertices of the graph. Edges may form the sides of more than one trapezoid, however; in fact they may be fragmented into as many as 2 log n segments, as stated above. Thus the search tree may have as many as O(n log n) nodes, which is the worst-case space complexity. The tree corresponding to the trapezoids found in Figure 4 is also shown.

The depth of the balanced search tree can be shown to be no more than $4\lceil \log n \rceil + 3$, so a search of the tree will take no more than that many steps. Thus the worst-case time complexity to locate a point in the planar partition is O(log n). Since there are O(n) edges and each edge may be segmented into O(log n) fragments, the time required to process the edges is O(n log n), while both the median-finding and the tree-balancing may be done in O(n log n) time. Thus the preprocessing required is O(n log n).

An added advantage which could be useful to our research is that the trapezoid method can be extended in some cases to finding a point among edges which are not straight-line segments. This can be done if first, the curves can be expressed as a single-valued function of one of the coordinates, and second, if it can be determined in constant time whether a point is on one side or the other of the curve.

5. Intersections Among Line Segments in the Cartesian Plane

A common operation of the algorithms proposed in Chapter VI is to find intersections between two piecewise-linear curves. It is thus important to find efficient methods of doing this operation. The intersection of two piecewise-linear curves with p and q segments respectively would take, using the naive approach which compares each segment of one curve with each segment of the other, O(pq) line segment intersections, so it is important to find better ways of doing the operation.

Preparata and Shamos present an algorithm to find all intersections among n line segments by performing a plane sweep along the x-axis. At any point on the x-axis, a vertical line imposes a total order on those line segments it intersects. This order is recorded in the sweep-line status. As the vertical line sweeps to the right from intersection point to intersection point, new line segments may be added to the ordering, and old ones deleted, but if any adjacent pair of line segments changes order, which is detected by a change in the

sweep-line status, an intersection of those segments must have occurred. Thus, any line segment which is added to the ordering is checked for intersection with the segment immediately above and below it by checking if the relative order changes at the point along the x-axis where the first of the two segments will be deleted. This approach can detect k intersections among n line segments where n is p+q in time O((n+k) log n). In our domain however, any two homogeneous-behavior-region boundaries will intersect in at most one point, because when any two boundaries intersect, a third boundary will begin and the other two will end. Therefore we could use a simplification of the above algorithm which will operate in O(n log n) time. [Ref. 8]

Intersection calculation for piecewise-linear curves with monotonic curvature can exploit these properties. Several algorithms of uncertain worst-case complexity seem to provide good empirical results. One in particular [Ref. 9] proceeds by constructing, in O(p) time, a bounding box for the first piecewise linear curve, and then checking, in O(q) time, which portion of the second curve, if any, intersects the bounding box. The intersecting portion of the second curve usually contains only a small fraction, call it k_1, of the whole curve, although it is at this point that the analysis becomes imprecise because k_i does not depend on p or q, but on the curvature and relative positions of the two curves. In any case, the next step is to reverse the roles of the two curves and create another bounding box about the k_1q line segments of the second curve, in $O(k_1q)$ time. The first curve is intersected with the new bounding box in O(p) time, finding k_2p segments which traverse the new box. The algorithm proceeds recursively as above, terminating when one of the bounding boxes contains only one line segment. At this point, the next check of the other curve will yield the exact intersection point. A rough approximation of the time complexity of this algorithm, if it is assumed for simplicity that at each stage the size of the curve is reduced by the same fraction k, is $T = ((1+k)q + 2p)/(1-k) + 1$. Thus this algorithm has, assuming $0<k<1$, time complexity $O(q+2p) = O(q+p) = O(n)$. This algorithm will not converge if at any stage the bounding box of each partial curve completely contains the other partial curve. But a simple check during each iteration to ensure that the sizes of the two curves are in fact decreasing will allow the method to proceed if it is converging. If it fails this test, a full O(pq) test of the two curves can be used instead.

6. Voronoi Diagrams

A technique in computational geometry that has been of use in some algorithms pertaining to optimal-path maps is *Voronoi diagram* construction [Ref. 8]. A Voronoi diagram Vor(S) with respect to a set of

points S in a plane is the partition of the plane such that each region contains the points with the same nearest neighbor in S. Figure 5 shows a typical Voronoi diagram. One method for constructing shortest-path maps (i.e., an optimal-path map for binary terrain), uses an extension of Voronoi-diagram methodology to plot approximations of the boundaries between homogeneous-behavior regions [Ref. 4]. It reduces the problem of constructing the planar partition to that of finding a Voronoi diagram for the vertices of an obstacle, where the costs of optimal paths from each vertex is known. Instead of bisectors between two vertices which are straight lines exactly half-way between them as described below, this method constructs bisectors which are either lines or hyperbola branches, depending on the nature of the paths from the two vertices. Then the Voronoi diagrams of single obstacles are merged to form the complete OPM.

Some observations about Voronoi diagrams lead to an initial construction method. Between two points P_1 and P_2 in the plane, the set of points closer to P_1 than to P_2 are the points in a half-plane containing P_1 defined by the perpendicular bisector of the line segment P_1P_2. Among a set S of n points in the plane, the set of points closer to a point P_i than to any other point in S is the intersection of n-1 half-planes each containing P_i defined by the perpendicular bisectors of the line segments P_iP_j. From this observation, a brute-force method of constructing a Voronoi diagram would be simply to construct each of the n polygons about each point in S. Since n half-planes can be intersected with each other in O(n log n) time by a divide-and-conquer approach, this approach takes time $O(n^2 \log n)$. [Ref. 8]

A more efficient approach for constructing Voronoi diagrams which also uses the divide-and-conquer paradigm can be summarized as follows. First, partition S into two sets S_1 and S_2 of roughly equal size according to whether the x-coordinate of each point is less than or greater than the median x-coordinate of points in S. Then, construct $Vor(S_1)$ and $Vor(S_2)$ recursively, and finally, merge $Vor(S_1)$ and $Vor(S_2)$ to obtain Vor(S). Partitioning S takes O(n) time for a set S of size n using a standard median-finding algorithm and the merging step takes O(n) time. If the entire algorithm can be performed in T(n) steps, the construction of both subordinate Voronoi diagrams in step two takes approximately 2T(n/2) time. So the recurrence relation $T(n) = 2T(n/2) + O(n)$ describes the algorithm, which when solved gives that T(n) is O(n log n).

The merging step is the heart of the algorithm, and is accomplished as follows. Because the map is partitioned such that S_1 and S_2 will lie on opposite sides of a vertical line, it can be shown that there is a chain

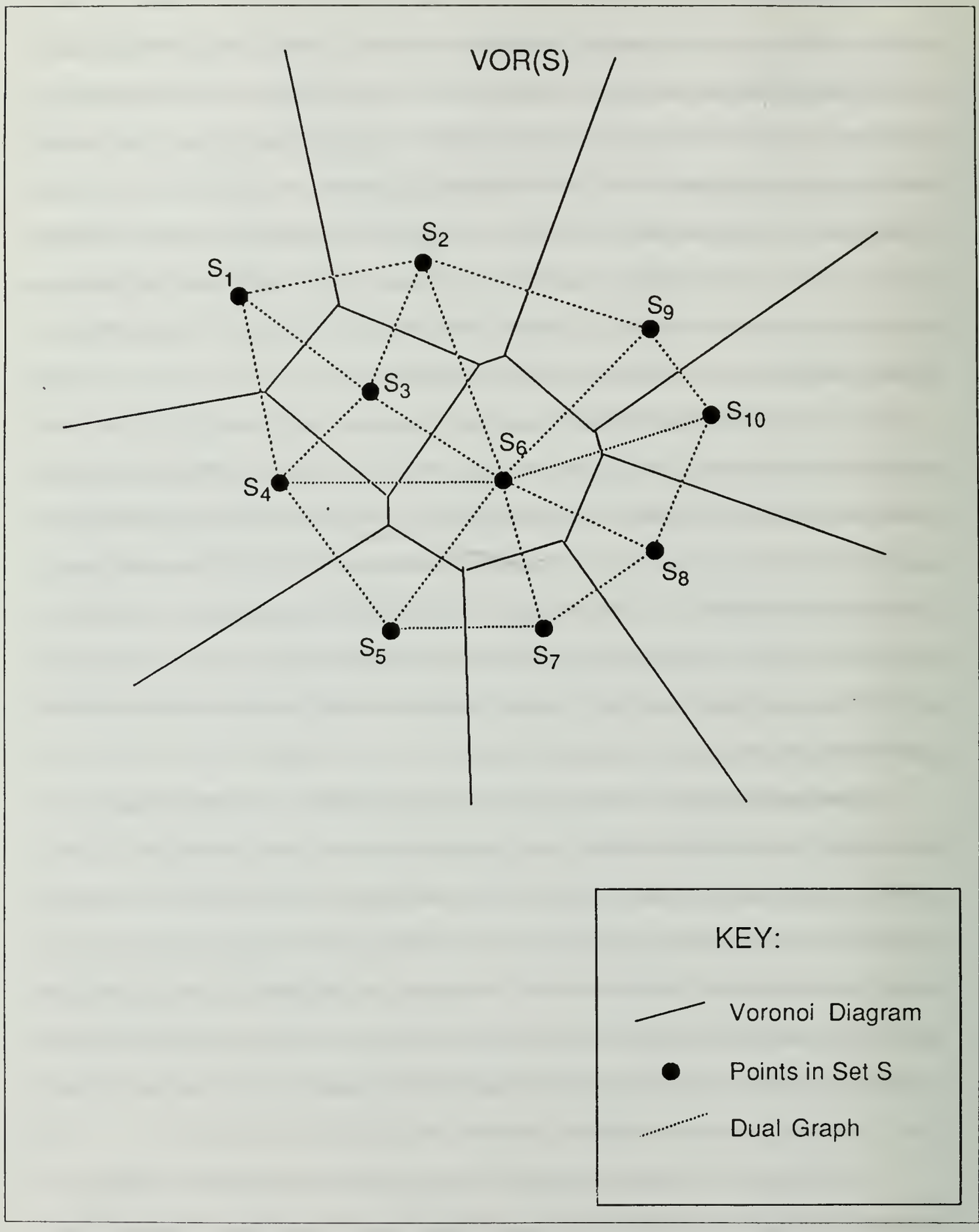


Figure 5

Voronoi Diagram

σ, i.e., a sequence of connected line segments, which is monotonic with respect to the y-axis, (i.e., no two points of the chain have the same y-coordinate) that also partitions the plane with S_1 and S_2 on opposite sides of σ such that the union of the portion of $Vor(S_1)$ which is left of σ and the portion of $Vor(S_2)$ which is right of σ yields Vor(S). In fact this chain σ can be constructed in linear time, so the recurrence relation stated above holds, and the construction of a Voronoi diagram can be done in O(n log n) time. [Ref. 8]

Generalizations of Voronoi diagrams have been presented which partition the plane into sets of points closest to a set of line segments [Ref. 10], or which base their distance function on metrics other than the Euclidean metric [Ref. 11]. Since OPMs for homogeneous-cost areas can be thought of as Voronoi diagrams with a different metric for each homogeneous-behavior region, the latter work seems promising. Currently, however, only several simple metrics such as L_1 and L_∞ have been considered, so more research in this area is necessary before OPMs of the type we are considering can be constructed with this approach. *Weighted Voronoi diagrams* [Ref. 12] assign a weight to each point about which regions are computed. This concept might appear useful in constructing OPMs, but is not. Instead it applies to a problem in which a mobile agent travels at different speeds depending on which terrain-feature vertex it just crossed.

What is needed in constructing OPMs for the binary case is a type of generalized Voronoi diagram in which the weight is the cost-rate of the region in which an obstacle vertex lies, and an offset of the initial weight at the vertex represents the cost of the optimal path from the vertex. This is, in essence, what the algorithm of Aronov [Ref. 13] computes. This algorithm allows points in the Voronoi set to be given an initial offset weight. Knowing that bisectors between such points are hyperbolas (or in the degenerate case, lines), they can be plotted just as in the basic Voronoi diagram algorithm. The key element of the method is the proof that a dividing chain can be constructed between two Voronoi diagrams as discussed above, which now can contain hyperbola segments as well as line segments. This allows smaller generalized Voronoi diagrams to be merged into larger ones, which is the foundation of the divide-and-conquer approach used.

C. DEFINITIONS OF RELEVANT FREE-SPACE PATH-PLANNING PROBLEMS

This thesis addresses problems where the mobile agent is of negligible size with respect to the surrounding terrain, where terrain is two-dimensional free space with fixed terrain features, where the environment is

stable and knowledge about it is complete, and where the optimality criterion is to minimize a cost function which is linear in path length.

A simplified version of this problem has been called by Lozano and Wesley [Ref. 1] and Brooks [Ref. 14] the FIND-PATH Problem, and by Mitchell [Ref. 15] the OBSTACLE-AVOIDANCE Problem. This simplified problem seeks any feasible path in terrain consisting of impassable obstacles on a homogeneous-cost background. An important extension to the FIND-PATH Problem includes the optimality criterion that the resulting path be the shortest among all feasible paths. It is called the OBSTACLE-AVOIDANCE SHORTEST-PATH Problem, or simply the SHORTEST-PATH Problem.

> **OBSTACLE-AVOIDANCE SHORTEST-PATH Problem**: Given a mobile agent A of negligible size with respect to the environment, an environment E consisting of impassable obstacles at fixed and known locations on a homogeneous-cost background, and motion objective O consisting of the translation of A to a specified goal point in the environment, find a continuous path π for A amidst E that achieves objective O such that its length is minimal among all feasible paths, or report that no feasible path exists.

Realistic terrain for large-scale cross-country path-planning can rarely be modelled as binary (i.e., obstacles on a homogeneous-cost background). A more useful assumption is that terrain can be modelled as homogeneous-cost regions. The map is consists of regions, each assigned a value representing the cost rate to the agent to traverse the region. The weighted-region problem is a generalization of the obstacle-avoidance shortest-path problem which defines terrain as homogeneous-cost regions.

> **WEIGHTED-REGION Problem:** Given a mobile agent A of negligible size with respect to environment E, E consisting of a partition of the plane into fixed homogeneous-cost regions of known position, and motion objective O consisting of the translation of A to a specified point in environment E, find a continuous path π for A amidst E that achieves objective O such that the path integral of the cost is minimal, or report that no feasible path exists.

The U.S. Army Engineer Waterways Experiment Station, the U.S. Army Engineer Topographic Laboratories (ETL) and the Defense Mapping Agency (DMA) currently can produce such cost-rate maps of environments E using a program called *Army Mobility Model* (AMM), for portions of the earth for which digitized terrain data is available. This data includes not only elevation data, but cultural, vegetation, and soil data as well, and must currently be collected in part manually [Ref. 16]. The output of AMM is a map in which terrain is subdivided according to the maximum speed with which the given vehicle could be expected to traverse the terrain.

If an application will require repeated solutions of the shortest-path or weighted-region problems, it may be more efficient to construct an *optimal-path-map* which represents optimal paths to a given goal point from all start points in the plane. If the output map represents solutions to the shortest-path problem, it is called a *shortest-path-map*. Some authors use *shortest-path-map* to refer to maps of the weighted-region problem as well, allowing the word *shortest* to mean shortest with respect to a specified cost function. We prefer the term *optimal-path-map*, however, to emphasize its basis in the weighted-region problem.

> **FIXED-GOAL OPTIMAL-PATH-MAP Problem:** Given mobile agent A of negligible size with respect to environment E, E consisting of a partition of the plane into fixed, homogeneous-cost regions of known position, and a set of motion objectives Θ which are to translate agent A from each of the continuum of start points S in the plane to a goal point G, represent the set Π of continuous paths for A in E that achieves objectives O_i in Θ such that the path integral for each π_i is minimal over all paths from start point S_i to G, or report that no feasible path exists.

D. TYPES OF PATH ERRORS

Several classes of errors may occur in algorithms which look for optimal paths. Each algorithm is based on a model of the path-planning domain with its own representation of reality, and operations manipulate that representation to produce a solution. For example, terrain in some models is represented by imposing a grid on the map and assigning a cost to each cell of the grid, while in some models terrain is represented by polygons with an assigned cost. Errors may occur either because of inaccuracies in operations within the model or because of inaccuracies in the model compared with the real-world domain. The first class below are errors of the former type, while the second and third classes are errors of the latter type.

1. Cost of Model Computed Path versus Cost of Model Optimal Path

Path-planning algorithms execute within the context of their model of real-world terrain. If an algorithm produces a solution path which has a computed cost greater than the minimum cost of some other path represented within the model, the algorithm has produced a model sub-optimal path. Such a solution may occur either intentionally or unintentionally. Some algorithms terminate when a candidate solution is guaranteed to be within some bound of the true model optimal solution, thus saving processing time at the expense of accuracy. An example of this type of algorithm is a variation of A* called A^*_ε [Ref. 17]. Another example of an algorithm which produces solutions with this kind of error is called simulated annealling. It uses stochastic methods to determine when a candidate solution has a high probability of being good enough [Ref. 18]. Er-

rors of this type also occur because of numerical errors in the mathematical operations performed by the algorithm. Standard numerical analysis techniques can be used to study these errors and attempt to reduce them.

2. Cost of Model Optimal Path versus Cost of Real-World Optimal Path

When the cost of the optimal path within the model is different from the actual cost of a path between the same two points in the real world, an error of the second class has occurred. Even the actual measurement of a path cost is only an approximation of reality, so any model produces at least some small error of this kind. The amount of this kind of error produced is an important consideration in choosing among algorithms. For example, as discussed in Section E below, the wavefront propagation algorithm may produce solutions which are optimal in its grid-based model, but which have as much as 7.6% greater cost than an actual path between the same two points as measured in the real world.

3. Location of Model Optimal Path versus Location of Real-World Optimal Path

A model optimal path could still be a valuable representation of a real-world optimal path despite a larger cost than the true optimal cost if its qualitative behavior was similar enough to the path it represented. But algorithms may produce solutions which follow quite different routes than the real-world optimal path. As discussed by Mitchell and Kiersey [Ref. 19], the grid-based model upon which wavefront propagation (see Section E below) is based allows for multiple paths with the model optimal cost, so only the details of the algorithm implementation determine which one is reported as the solution, and that reported solution may differ markedly from the true optimal path. This type of error may or may not be important depending on the application to which the results will be applied.

E. RELEVANT OPTIMAL-PATH PLANNING RESEARCH

A taxonomy for categorizing free-space path-planning methods is presented in Figure 6. Algorithms for free-space path planning generally transform an infinite search space into a finite one by eliminating all but a finite number of candidate paths, and then searching this finite space using standard techniques such as branch-and-bound or A* search. Two distinct ways used to effect this transformation to a finite search space are map discretization and spatial reasoning.

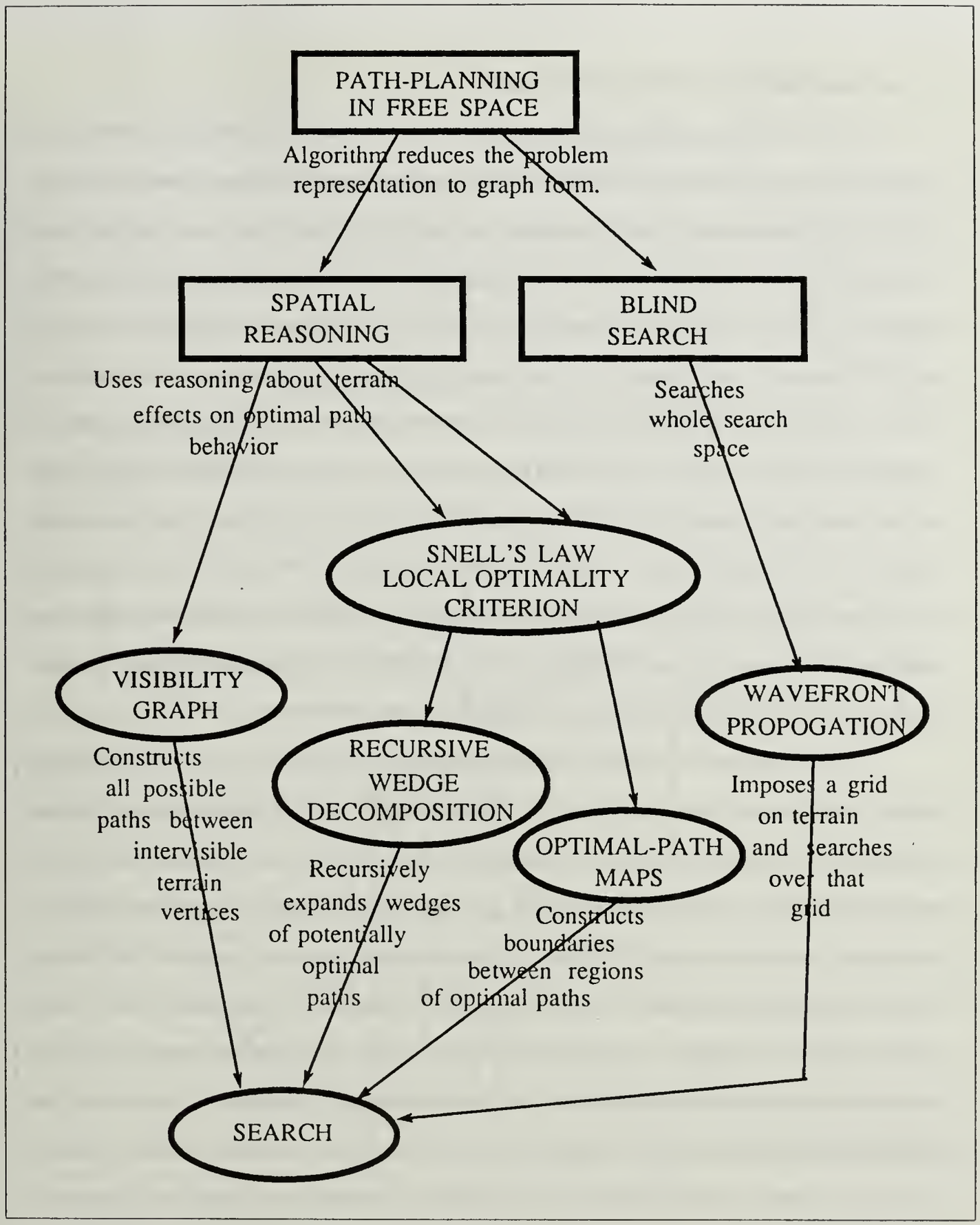

PATH-PLANNING IN FREE SPACE
Algorithm reduces the problem representation to graph form.
SPATIAL REASONING
BLIND SEARCH
Uses reasoning about terrain effects on optimal path behavior
Searches whole search space
SNELL'S LAW LOCAL OPTIMALITY CRITERION
VISIBILITY GRAPH
RECURSIVE WEDGE DECOMPOSITION
OPTIMAL-PATH MAPS
WAVEFRONT PROPOGATION
Constructs all possible paths between intervisible terrain vertices
Recursively expands wedges of potentially optimal paths
Constructs boundaries between regions of optimal paths
Imposes a grid on terrain and searches over that grid
SEARCH

Figure 6

Taxonomy of Free-Space Path-Planning

1. Map Discretization - Wavefront Propagation

Map discretization methods approximate the terrain by imposing a tesselation on the map and categorizing each cell according to the terrain it overlays, and allow travel only between centers of grid cells. Alternate representations are possible, for example, where travel is allowed between corners of cells. Since there are a finite number of cells, there are a finite, though large, number of candidate paths (assuming cycling is prevented). A method popular for its simplicity is called *wavefront propagation* (see Figure 7) [Ref. 15], [Ref. 20]. The terrain is approximated by a square tesselation of the map, and paths are approximated by allowing motion only from the center of a cell to the center of an adjacent cell. Eight-neighbor adjacency is usually used, meaning that from a cell, the agent may move to any of the four perpendicularly adjacent cells or to any of the four diagonally adjacent cells. Because of the restrictions on directions of movement, eight-neighbor wavefront propagation has as much as 7.6% inaccuracy in that a reported solution may cost as much as 7.6% more than the real-world optimal path [Ref. 20]. Normally Dijkstra's algorithm (branch-and-bound search) is used to expand in all directions from the start point until the goal is first reached. The name wavefront propagation is used because of the analogy of the expansion of a circular wave in water.

The implementation of wavefront propagation reported by Richbourg [Ref. 21] is a variation of Dijkstra's algorithm which models the expansion of the wavefront explicitly. The basic mechanism is that time is incremented in fixed units, and at each time increment the wavefront is propagated outward as far as it can travel through each cell currently on the wavefront. Each cell which is reached by the wavefront is added to the wavefront list, and when the cell's cost has been decremented below zero it is dropped off the wavefront list. During each iteration, cells through which the wavefront has fully passed will propagate the wave to each of their neighbors. If the neighbor cell has not yet been reached by the wavefront a back-pointer is set back to the cell on the wavefront and the neighbor cell's cost is decremented according to how far the wavefront can travel through it in a unit of time. If the neighbor cell has already been reached by another cell on the wavefront, no action will be taken unless the neighbor cell's cost could be decremented further by the currently propagating cell than it was decremented by the previous cell. In that case, the pointer is changed to point to the current cell and the cost is set accordingly.

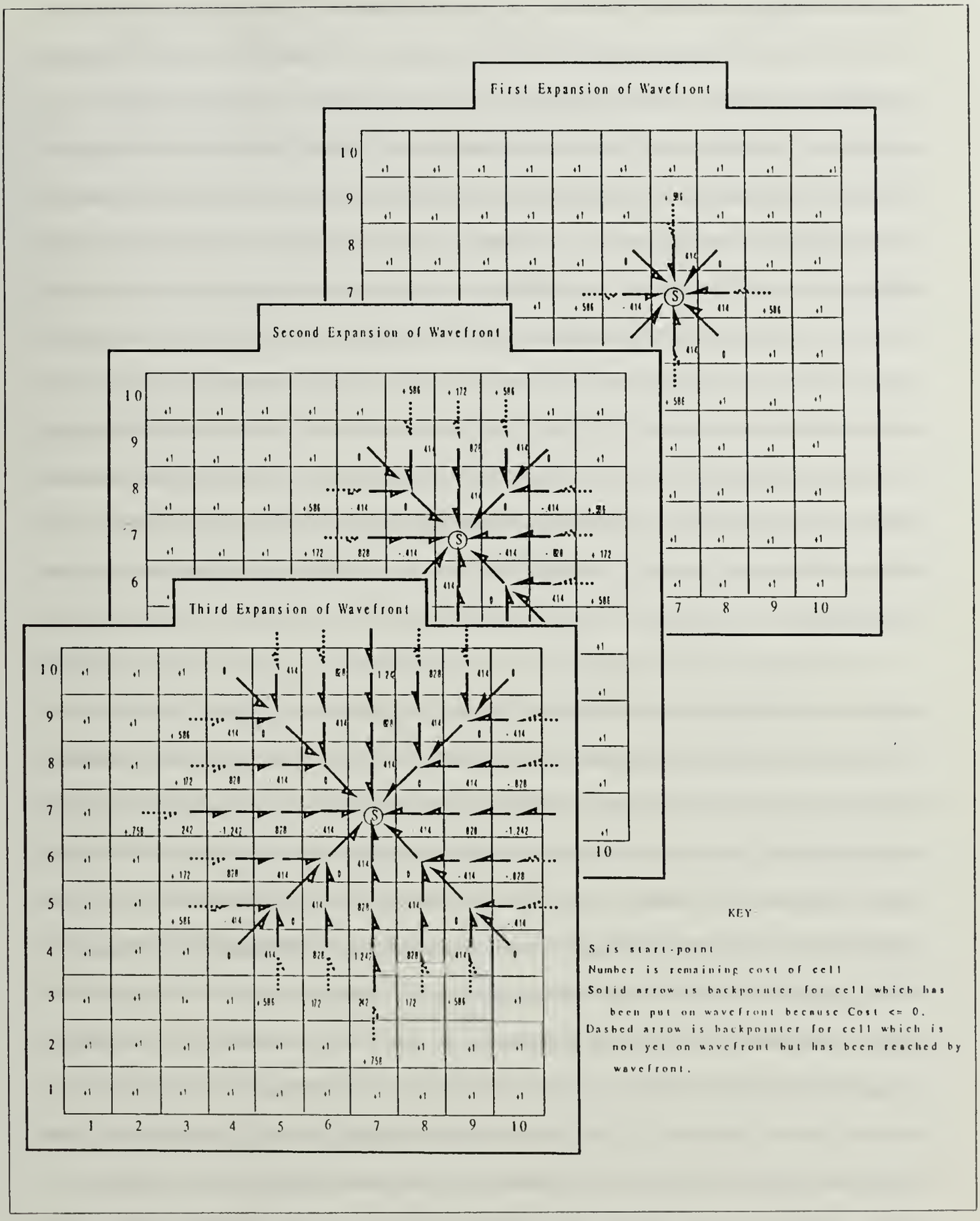

First Expansion of Wavefront
Second Expansion of Wavefront
Third Expansion of Wavefront
KEY
S is start-point
Number is remaining cost of cell
Solid arrow is backpointer for cell which has been put on wavefront because Cost <= 0.
Dashed arrow is backpointer for cell which is not yet on wavefront but has been reached by wavefront.

Figure 7

Wavefront Propagation

If cells are square and have unit dimensions, time could be advanced in increments such that it would take 1 time unit for the wave to pass through a cell of unit cost in an orthogonal direction and $\sqrt{2}$ time units in a diagonal direction. For a cell with a cost of c, the wave will take c and $c \cdot \sqrt{2}$ time units respectively. Alternately, we will adopt the convention that time is incremented in units of $\sqrt{2}$, so that the wave will progress $\sqrt{2}/c$ units of distance through a cell of cost c in the orthogonal direction in one iteration, and 1/c distance in the diagonal direction in one iteration. This convention provides that, for cells of integral cost, diagonal expansion of the wave will always end inside the cell or at its edge, never overflowing into the next cell, so it is only in the orthogonal direction that it is necessary to check for overflow. Thus we decrement the original cost associated with a cell by 1 or by $\sqrt{2}$ at each iteration, and when the remaining cost is less than zero, we know that the wave has passed completely through it. Figure 7 illustrates the mechanics of the wavefront propagation algorithm. The figure shows a sequence of snapshots of the algorithm, where the remaining cost of each cell is noted inside the cell, and arrows represent pointers to each parent cell. The arrows are solid when the cell has been added to the wavefront, and dotted when the cell is not yet on the wavefront but has a back-pointer assigned.

For a map of m cells, the worst-case time complexity of Dijkstra's algorithm is O(m log m), [Ref. 15], or if we consider the two-dimensional nature of the input map, say of size n by n cells, the complexity is $O(n^2 \log n)$. This version does not depend on the costs of cells on the map. But for the version used by Richbourg, time complexity is also a function of initial costs of the cells. Each cell will remain on the wavefront until its initial cost c is decremented below zero. The cell's cost will be decremented by 1 or $\sqrt{2}$ at each iteration, so each cell will remain on the wavefront for O(c) iterations. Each iteration that a cell is on the wavefront, its eight neighbors will be checked to see if the shortest path yet to the neighbor cell is through the cell being considered, or through some already-processed cell. Thus in the worst-case where all cells have a cost c_{max}, if we assume that there is some upper bound on the cost of cells, the worst-case time complexity is $O(c_{max} m)$. In terms of an n by n input map, the worst-case time complexity is $O(c_{max} n^2)$. We must for theoretical reasons make the assumption that there is an upper bound on the magnitude of c_{max}, because if c_{max} is unbounded, and as usual is represented in $\log c_{max}$ bits, we have that the size of the input map is $I = O(\log c_{max})$, or $2^I = c_{max}$. Thus the worst-case time complexity would be $O(2^I m)$. As explained in [Ref. 22], this type of algorithm has pseudo-polynomial time complexity, i.e., it is polynomial if the input size is bounded, but ex-

ponential if the input size is allowed to be unbounded. Both versions of wavefront propagation have space complexity of O(m).

Dijkstra's algorithm examines paths in all directions from the start point, regardless of which are likely to lead to the goal point quickest. But extending the algorithm to A* search by introducing an evaluation function gives large increases in execution speed by focusing the search on paths which seem to be proceeding in the best directions. The evaluation function used in the A* version of wavefront propagation measures the Euclidean distance from the cell currently being considered to the goal cell. Mitchell and Kiersey [Ref. 19] report an increase in speed for A* search over Dijkstra's algorithm of 1.5 to 20 times.

Increased resolution of the tesselation will not reduce the worst-case inaccuracy of reported solutions below the 7.6% upper bound. This inaccuracy, called *digitization bias*, arises because of the discrete approximation of paths. The only way to reduce the upper bound on error caused by digitization bias is to increase the number of possible directions the mobile agent is allowed to travel. Sixteen-neighbor wavefront propagation, for example, allows paths between a cell and the sixteen cells which are separated from it by one cell. Richbourg [Ref. 20] showed how sixteen-neighbor adjacency could decrease the inaccuracy to approximately 1.9%.

Not only does digitization bias lead to inaccuracy, it also means that multiple solution paths could be reported depending on implementation details of the algorithm. Path representations approximate the true optimal path in the actual terrain by connected line segments which lie in allowed directions. So a true optimal path which for example lies at a 22.5° angle with the horizontal could be represented by one which starts in a 45° direction, and then finishes in a horizontal direction, or it could be represented by one which alternates many times between small 45° line segments and horizontal line segments, somewhat like computer graphics routines represent lines with sets of pixels. The latter representation is to be preferred because it more closely approximates the true optimal path, and some researchers have proposed ways to augment wavefront propagation algorithms to favor paths which have more regular turns, so as to better approximate line segments. [Ref. 19], [Ref. 20], [Ref. 23], [Ref. 24], [Ref. 25]

Mitchell and Kiersey [Ref. 19] discuss an implementation of wavefront propagation called BITPATH which partially compensates for digitization bias by modifying the way in which path distances are computed. Vossepoel and Smeulders [Ref. 26] developed an estimate for the actual distance over a true optimal path given

a digitized approximation which lowers the estimate each time the approximation path turns, based on the idea that each turn point suggests overestimation of Euclidean distance. BITPATH incorporates this estimate as the cost function of A*, i.e., the value assigned to a cell to represent the cost of the best path from the start cell. They claim a significant improvement in BITPATH's ability to find a solution which not only has minimum cost of all possible paths, but also lies close to the true optimal path. [Ref. 19]

In an attempt to reduce the dependency of accuracy on resolution, data representation schemes that use multiple resolutions have been introduced which use hierarchical algorithms which are generalizations of wavefront propagation [Ref. 27]. One such scheme uses quad-trees to represent larger homogeneous areas with single cells [Ref. 28]. With this approach, rectangles are inscribed within homogeneous-cost regions of the input map, and then successively smaller rectangles fill out the shape of the regions. This representation is then searched much the same as in wavefront propagation.

A parallel processing approach to wavefront-propagation path planning has been implemented in support of the DARPA-sponsored autonomous land vehicle built by Martin Marietta [Ref. 29]. Multiple processors are utilized to sweep horizontal bands of the map, at each cell replacing the current cost of its neighbors if the current cost of the cell plus the cost to move to the neighbor is less than the neighbor's current cost. Multiple sweeps are employed until the cost values stabilize. Richbourg [Ref. 20] suggests an alternative based on mesh-connected architectures in which computational elements in the architecture would model cells in the map, yielding an O(n) algorithm, and Jorgenson [Ref. 30] presents a wavefront propagation implementation on a neural-network machine.

2. Spatial Reasoning Methods

Spatial reasoning uses principles about how optimal paths must behave in the presence of terrain features to constrain the search space for optimal paths. A simple example of this type of reasoning is that optimal paths are always straight lines across homogeneous terrain, and in the case of binary terrain (obstacles on a homogeneous-cost background), turn only at obstacle vertices (see Theorem I-2, Appendix A). A more general type of discretization than that used by wavefront propagation takes place when terrain features are modelled using polygons. Here, error in model optimal paths versus real-world optimal paths can be much less than with rectangular tesselations, but since algorithms which use this type of discretization have com-

plexities which depend on the number of terrain-feature vertices in the map, there is a trade-off between accuracy of representation and speed of execution.

Path-planning methods have used at least four distinct techniques which can be considered spatial reasoning techniques, with many algorithms appealing to more than one of the techniques. They are visibility-graph methods, the Snell's Law local optimization criterion, the continuous-Dijkstra paradigm, and methods using optimal-path maps.

a. Visibility Graphs

Visibility-Graph methods [Ref. 1] solve the polygonal obstacle-avoidance shortest-path problem (binary terrain), constructing a graph where each of the n obstacle vertices plus the start and goal points are nodes, and undirected arcs connect nodes whose vertices are intervisible, i.e., can be connected by a line segment which does not intersect any obstacle edge. Because of the spatial reasoning principle about binary terrain stated above, it is assured that every segment of an optimal path will occur in the visibility graph, so to find an optimal path it is sufficient to search the graph using branch-and- bound search.

Several algorithms have been given to construct the visibility graph. The naive algorithm checks every pair of vertices against every edge to see if the line segment connecting them intersects the edge. Since there are $O(n^2)$ pairs of vertices and O(n) edges, this brute force algorithm has worst-case time complexity $O(n^3)$. Lee [Ref. 31] and Mitchell [Ref. 32] explain an $O(n^2 \log n)$ algorithm which begins by constructing for each vertex a list of the other vertices sorted according to the heading of the line between them in $O(n^2 \log n)$ time, and then for each of the n sorted sets, doing an angular sweep checking for intersection against the closest obstacle edge. Welzl [Ref. 33] and Asano [Ref. 34] used the fact that n sorts can be done in $O(n^2)$ time to build an $O(n^2)$ visibility graph construction algorithm. Ghosh and Mount [Ref. 35] give an algorithm to compute the visibility graph of n disjoint line segments in time $O(e + n \log n)$, where e is the number of edges in the visibility graph (an output-sensitive complexity). Since e may be as small as n or as large as n^2, this algorithm's worst-case time complexity ranges from $O(n \log n)$ to $O(n^2)$ depending on the size of the visibility graph.

Once the visibility graph has been constructed, Dijkstra's algorithm or the special case of it called A* (see Section A), may be used to search for the shortest path from the start to the goal. The worst-case time complexity of Dijkstra's algorithm is given by Aho, Hopcroft, and Ullman as $O(e \log n)$ [Ref. 36]. Again, because of the range of e, this means that Dijkstra's algorithm is, in the worst case, $O(n^2 \log n)$, or with a sparse

visibility graph, O(n log n). A*, an "informed" version of Dijkstra's algorithm, has time complexity of the same order class in the worst case [Ref. 15], although actual implementations should show a significant empirical superiority of A*. Thus, the shortest-path problem can be solved by a visibility-graph approach in $O(n^2 \log n)$ time.

For the variation of the weighted-region problem (or generalization of the shortest-path problem) given by Rowe [Ref. 2] which considers roads and rivers as well as obstacles, a visibility-graph-influenced approach is used to transform the search space to a finite one. Reasoning about how optimal paths must behave in the presence of roads and rivers leads to the conclusions that a path will enter or leave a road at only one critical angle, and that paths either cross a river without changing heading, or go around river-end vertices as they would an obstacle vertex. A visibility graph is constructed using as nodes all obstacle and river vertices and start and goal points; roads and rivers are not considered to obscure visibility. Additionally, line segments from each node are constructed which intersect each road at the critical angle. If the points are otherwise visible, the road-intersection point is added as a node and the graph reflects that the points are connected. Further, all nodes which lie on contiguous road segments are connected. This graph is then searched using Dijkstra's or A* algorithms as above. Figure 8 shows the edges of an example generalized visibility graph. In this figure, solid lines represent roads, dotted lines represent rivers, and filled polygons represent obstacles. Narrow dashed lines represent V-graph edges and the thick dashed line represents the optimal path from start to goal points. Similar results for linear features are reported by Gewali et al. [Ref. 37]

b. Snell's Law Local Optimality Criterion

Optimal paths in the weighted-region domain obey an analogy to *Snell's Law of Refraction* in optics [Ref. 20], [Ref. 3], [Ref 38]. Snell's Law is based on *Fermat's Principle* which says that light seeks the path of minimum time. Fermat's Principle has an analogy in the weighted-region problem, since time is a cost proportional to distance travelled in a homogeneous-index region. Thus optimal paths follow Snell's Law.

> **Snell's Law for Optimal Paths**: An optimal path passing through an edge between two regions with costs-per-unit-distance c_1 and c_2 obeys the relationship $c_1 \sin \theta_1 = c_2 \sin \theta_2$, where θ_1 and θ_2 are the angles of incidence and refraction respectively, i.e., the angle from the path in the first region to a line normal to the edge, and the angle from the path in the second region to a line normal to the edge. (See Figure 9.)

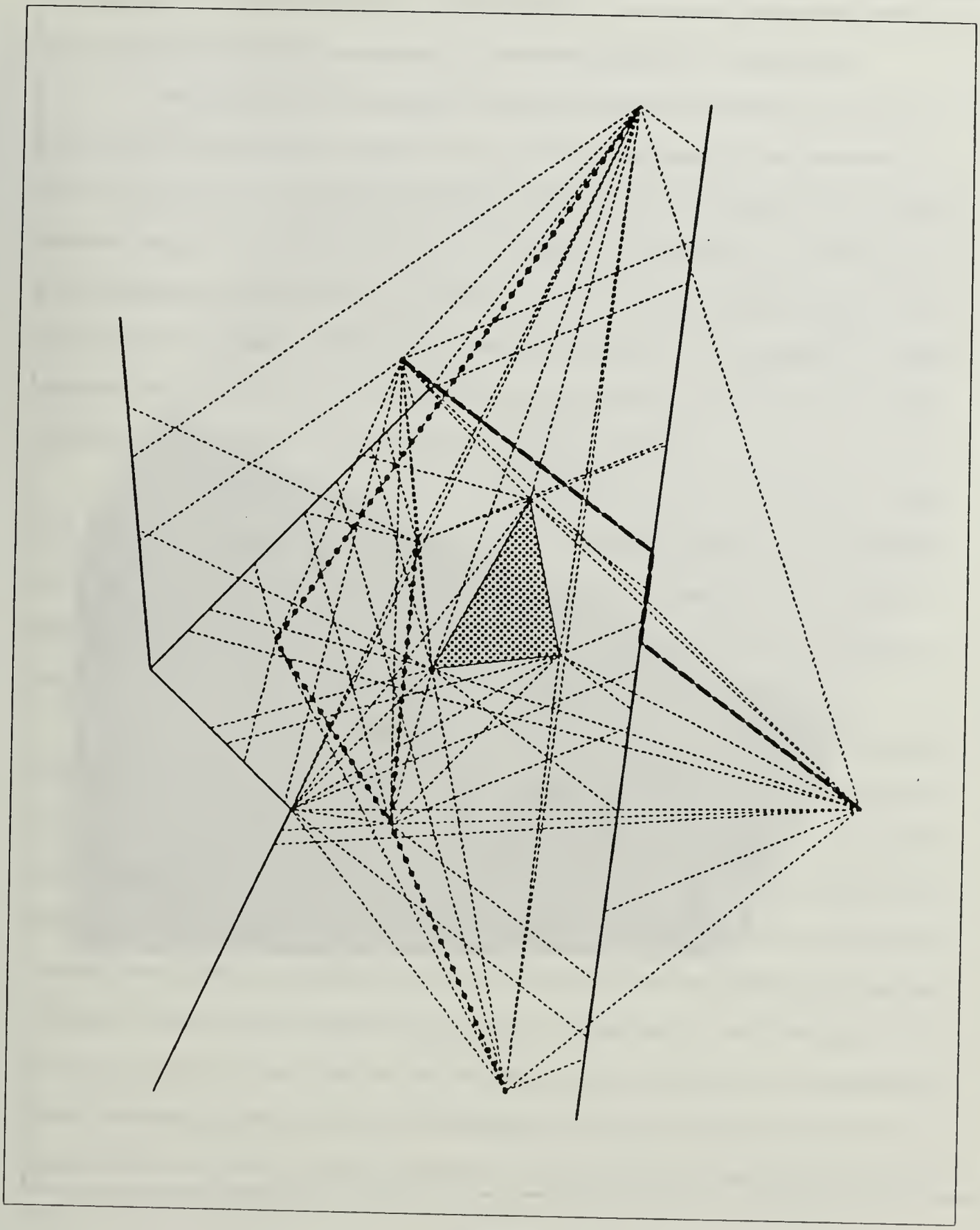


Figure 8

Visibility Graph for RRR Algorithm

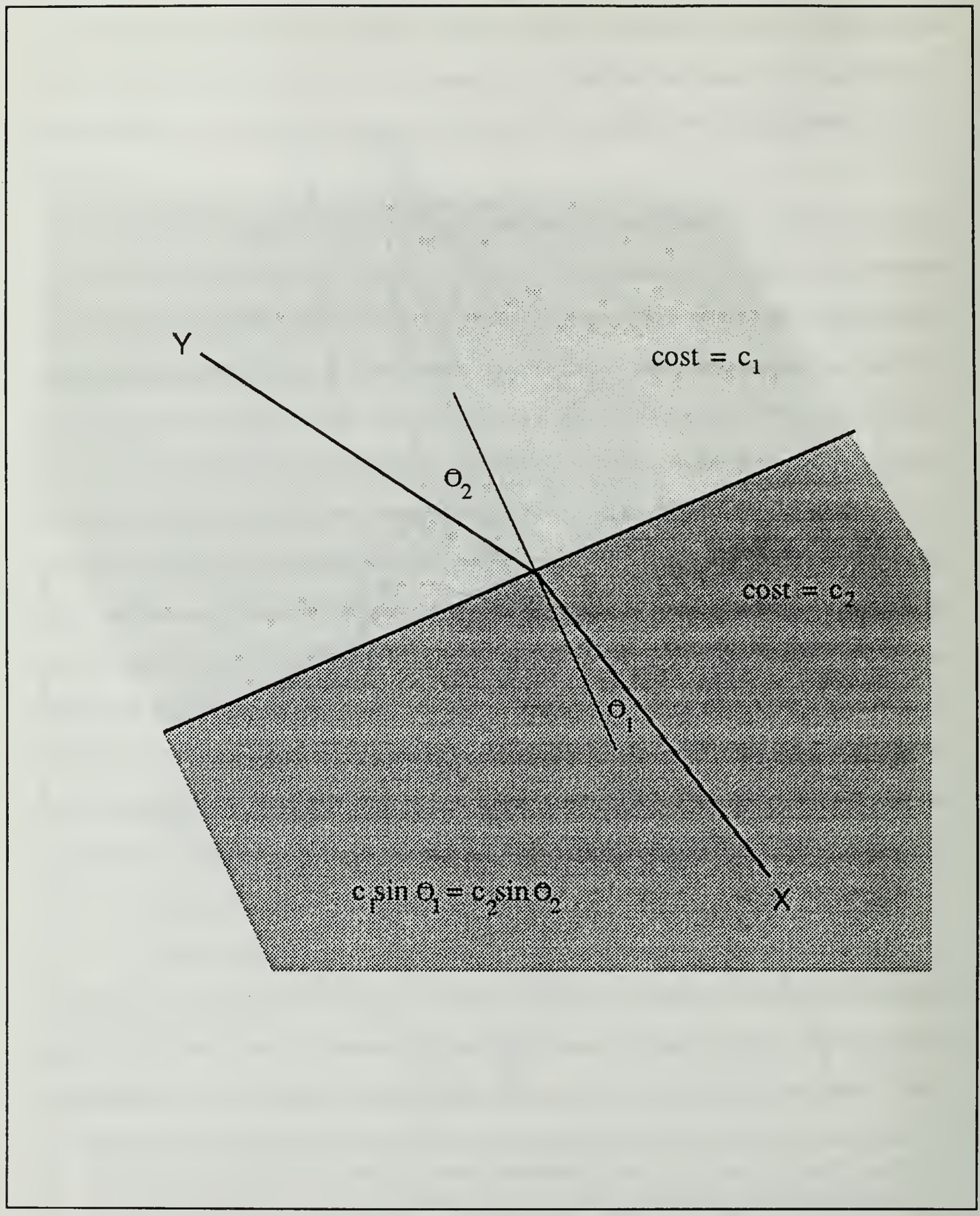


Figure 9

Snell's Law for Optimal Paths

Note that Snell's Law is a criterion for local, not global, optimality; a non-optimal path may obey Snell's Law at each edge crossing. Therefore, its usefulness is in constructing candidates for global optimality.

The analogy to Snell's Law applies to crossings with an angle of incidence and refraction such that θ_1 and θ_2 are both less than or equal to 90°. In the path-planning domain optimal paths cannot occur that have angles of incidence greater than a *critical angle* which is $\theta_c = \sin^{-1} c_i/c_j$, where $c_i < c_j$ and c_j is the cost of the region on the incidence side of the edge. For example, in Figure 10 an optimal path may go from point S to any point to the left of point A, but may not go immediately to its right, because the angle θ that line AB would form with edge PQ of the high-cost region would exceed θ_c. This is called total internal reflection, in optics. Another example of such behavior is found in Figure 11, where a path SABG follows Snell's Law making an angle exactly the critical angle at point A and then at B.

In Figure 12, paths just to the left of SV will be refracted according to Snell's Law as is path SVA, while paths just to the right of SV will be refracted as is path SVB, but paths which go through point V may lie anywhere within the wedge formed by AVB. If we consider that the edges which meet at point V are actually continuously curved there, Snell's Law will apply as the local curvature increases to infinity. The same behavior happens in Figure 13, at vertex V of an obstacle.

Finding an exact Snell's-Law path between two points through a sequence of edges requires an iterative search. Richbourg [Ref. 20] and Mitchell [Ref. 15] both discuss the lack of a closed-form solution for the problem of finding the Snell's-Law path between two points. But since it is an easy task to trace a Snell's-Law path from a point with a given heading, both conclude that an iterative search is the best approach. Richbourg studies the effectiveness of four techniques for finding, to within a given error, a Snell's-Law path across one edge. He used experiments applied to bisection search, golden-section search, false-position search, and a modification called heuristic false-position search, and found that the latter converged more than twice as fast as any of the others on the average, and also had the least standard deviation of the four methods. His heuristic false-position method attempts to avoid the situation where the search approaches the solution from the same side at each iteration, since false-position tends to converge more quickly when the solution is bracketed.

Mitchell's algorithm uses a numerical routine to approximate the Snell's-Law path across n edges which is of a time complexity that he calculates is bounded by $O(n^4L)$, where L is a measure of the precision

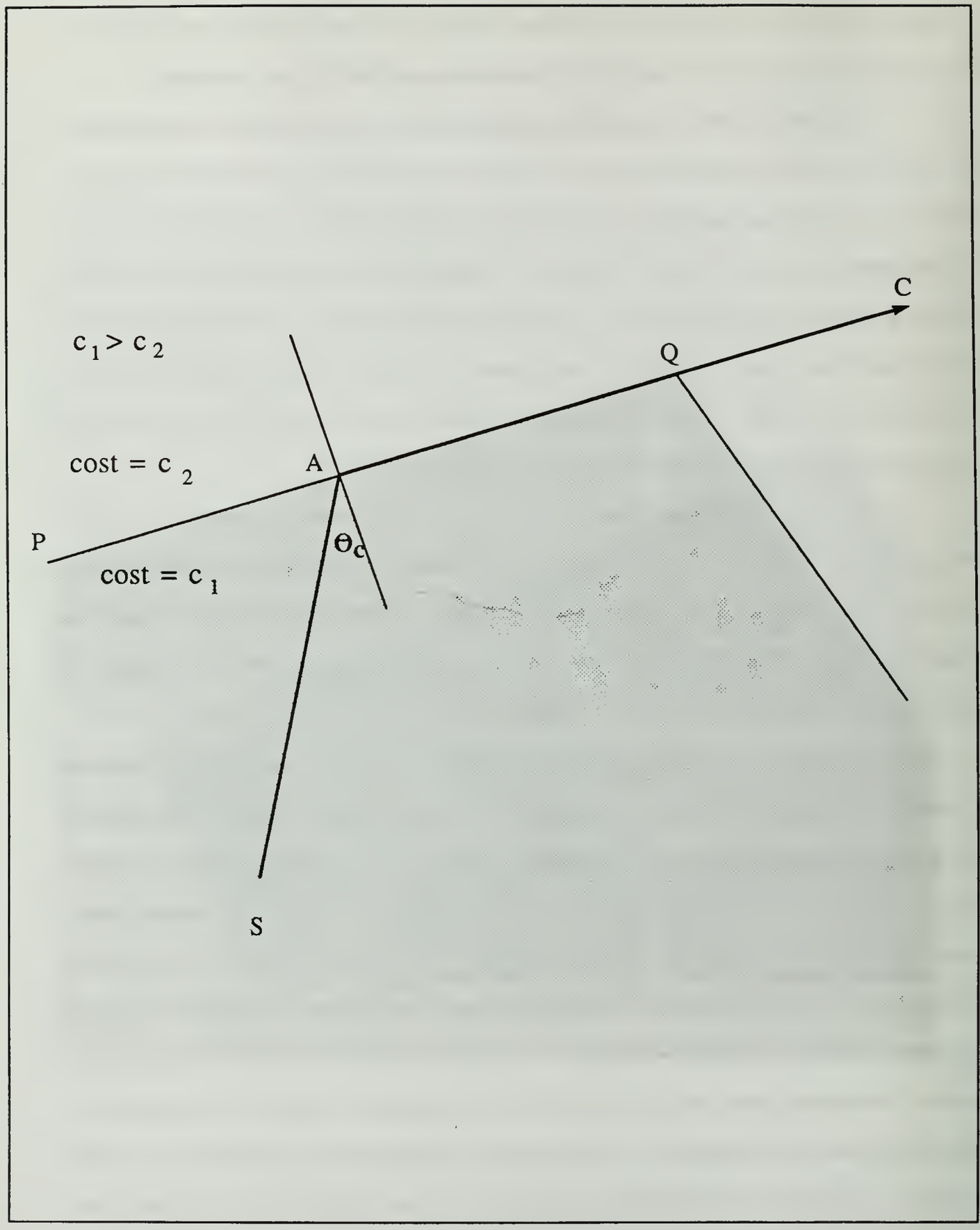


Figure 10

Snell's Law Example 1

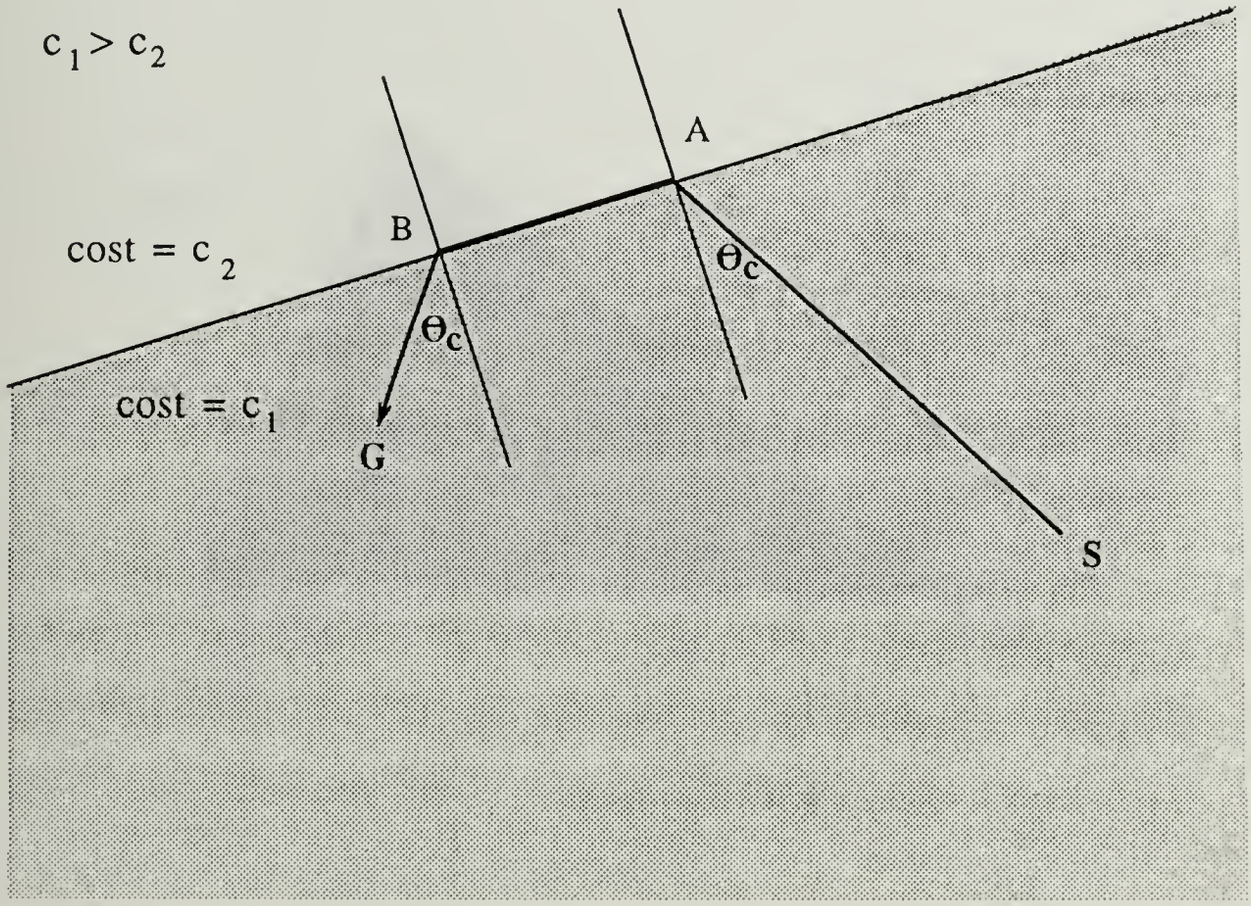


Figure 11

Snell's Law Example 2

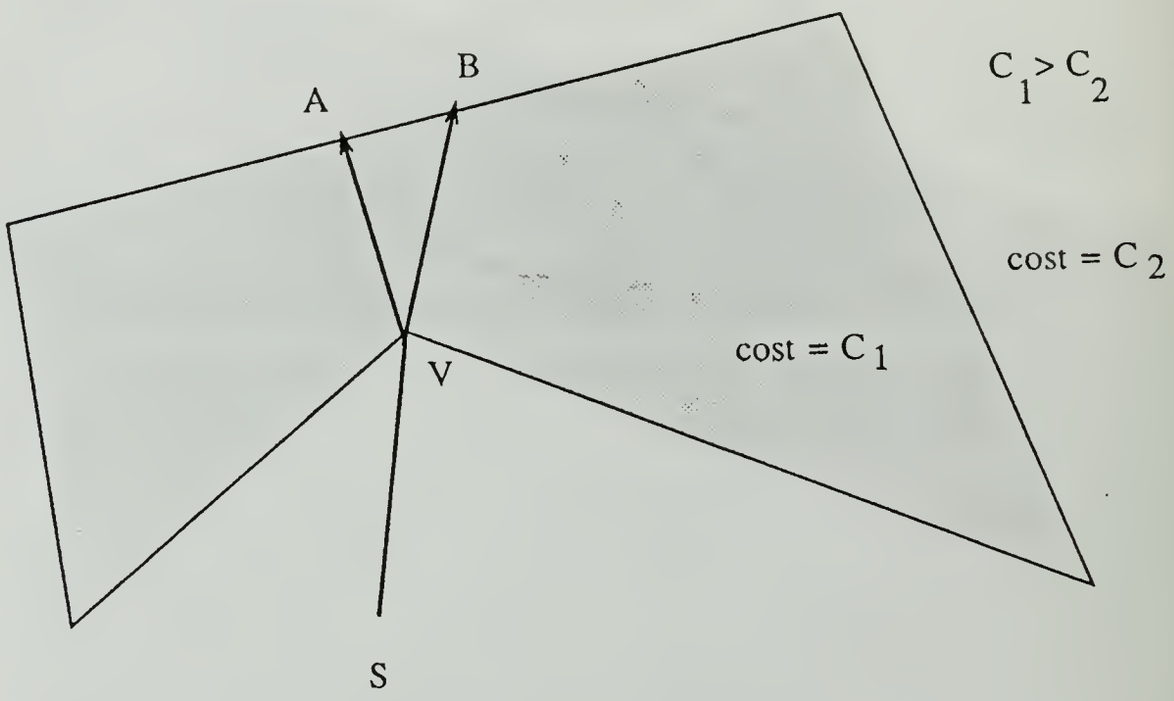


Figure 12

Snell's Law Example 3

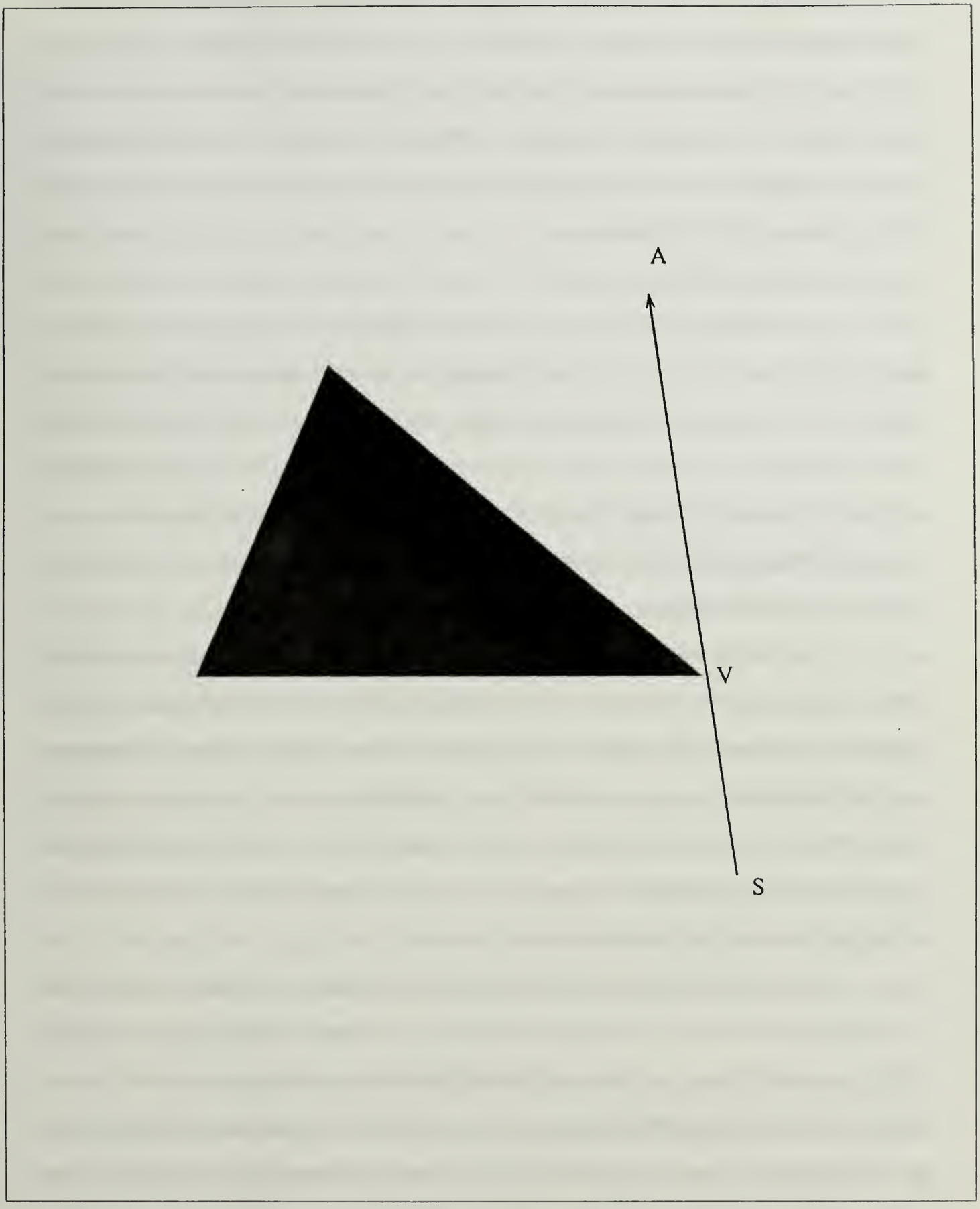


Figure 13

Snell's Law Example 4

of the problem instance. This routine is used because of its proven worst-case speed, but he also reports a coordinate descent method for solving a Snell's-Law path across several edges which is said to have fast empirical convergence. This method uses as a starting path a sequence of line segments between start and goal points through the midpoint of each edge. It then successively adjusts each crossing point in constant time to satisfy Snell's Law with respect to its two neighbor crossing points, iteratively applying these adjustments until the path is within a specified error of the goal.

c. Continuous-Dijkstra Paradigm

Path-planning methods for the weighted-region problem have used one of two similar approaches, both relying on Snell's Law and related properties as discussed above. Mitchell's algorithm uses what he calls the *continuous-Dijkstra* paradigm, because of its analogy to the discrete Dijkstra algorithm [Ref. 3], while Richbourg's algorithm uses *recursive wedge decomposition* [Ref. 20], [Ref. 21]. Whereas Dijkstra's algorithm must be used over terrain approximated by map discretization, the continuous-Dijkstra and the recursive-wedge-decomposition paradigms are used over terrain in which terrain features are represented by polygons or piecewise-linear curves.

The continuous-Dijkstra paradigm, analogously to searching a finite graph for the next closest node in Dijkstra's algorithm, searches in a concentric plane sweep outward from the start point, processing each terrain-feature vertex as the sweep reaches it. The algorithm requires the triangulation of the terrain map, a task for which standard algorithms are available from computational geometry. Each vertex has associated with it a label which represents the cost of the best path yet found to it, just as in the discrete Dijkstra algorithm. Additional points, called *frontier points*, also have labels associated with them. They are points in the interior of an edge at which critical reflection occurs (see above).

The key data structures for Mitchell's algorithm are first, a list of subsegments of terrain-feature edges called *candidate intervals of optimality*, and second, a priority queue called the *event queue* after the terminology used in the plane sweep paradigm. Candidate intervals of optimality are the extent of an edge over which an optimal path could possibly lie by the constraints of Snell's Law. Intervals include information about the *root*, or last previous vertex through which the all optimal paths which cross the interval lie, and about the paths from this root to either end of the subsegment interval. The event queue contains those points which are end points of some candidate interval, or are frontier points in the interior of an interval.

At each step of the algorithm, the point on the event queue with the smallest cumulative cost from the start point is chosen. If it is a frontier point, then the candidate interval is said to *propagate*. In other words, more candidate intervals on other edges are found each of which includes an edge subsegment to which optimal paths could arrive through the initial interval. The appropriate points have their costs computed and are added to the event queue. When the event queue becomes empty, the algorithm terminates, and the goal point has been labelled with its optimal cost. The list of candidate intervals holds, at any point in the algorithm, the best path or set of paths so far from the start point to the interval, so the interval which is the goal point is found in order to retrieve the optimal path. This algorithm has at most $O(n^3)$ event points, and uses the $O(n^4L)$ routine discussed above to find a Snell's-Law path between two points, and so has a worst-case time complexity of $O(n^7L)$, where n is the number of terrain-feature vertices and L is a measure of the precision of the problem instance.

Richbourg's algorithm uses A* search to select a group of paths for refinement which offers the best hope of containing the optimal path from the start point to the goal [Ref. 20]. As refined Rowe and Richbourg [Ref. 39], a *well-behaved path subspace* (WBPS) is defined as a set of paths which cross the same terrain- feature edges and vertices from the start to the goal. A *wedge* is a partial WBPS which is a set of paths crossing the same edges and vertices from the start point to some intermediate point or edge. *Refining* a wedge means finding within the wedge the nearest intermediate point which has not yet been considered, finding a Snell's-Law path to that point, and splitting the wedge into three sub-wedges based on the cases which arise from Snell's Law. These three wedges are added back to the A* agenda for further consideration. Two of the three wedges are those consisting of paths which pass to the "left" and "right" of the point at which splitting occurs, while the middle wedge is constructed based on the possible behavior of paths emanating from the point. The term recursive wedge decomposition refers to the successive splitting of wedges as they are selected from the A* agenda and refined.

The search space for recursive wedge decomposition is a known feasible start-to-goal path and a set of wedges with associated lower-bound values of cost function plus evaluation function for each wedge. These lower-bound values represent the lowest possible cost for a path within the wedge. The known feasible path is replaced whenever a better path is found, so that it is always the best known path. The single operator for state transformation is wedge refinement. The algorithm uses a different termination criterion than that

normally used by A* in path-planning applications. Normally the search can stop when the first element on the agenda is a complete solution, because the agenda is ordered by increasing cost-function plus evaluation-function values, and for a complete path the evaluation function equals zero and the cost function is the actual cost from start to goal. However, in this search space, the elements on the agenda are wedges, not paths. The search terminates when the best wedge on the agenda (and hence all other wedges on the agenda as well) has a cost-function plus evaluation-function value that exceeds the upper-bound cost of the current best feasible known path, or when the agenda is empty. In either case the least-cost known path is the solution. Wedges are pruned, or removed from the search space, according to a set of criteria based on Snell's Law and other spatial reasoning. An implementation of Richbourg's recursive-wedge-decomposition algorithm is reported to have empirical performance which strongly suggests an $O(n^2)$ average-case time complexity, where n is the number of terrain-feature vertices. Worst-case time complexity was reported to be $O(n!n^2)$ [Ref. 39].

The two algorithms are quite similar in some respects. The candidate interval of the continuous-Dijkstra algorithm along with its associated data about boundary paths corresponds to the wedges of the recursive-wedge-decomposition algorithm, and propagation of intervals corresponds to refinement of wedges. The same properties of Snell's Law refraction and critical reflection are used in determining how to refine wedges (propagate intervals). However, there are differences of emphasis. The focus of the continuous-Dijkstra algorithm seems to be finding a polynomial-time worst-case algorithm, while the A* search of the recursive-wedge-decomposition algorithm focuses on average-case performance. The continuous-Dijkstra algorithm requires a triangulation of the input map, a time-consuming preprocessing step which nevertheless does not raise the worst-case time order of complexity, while the recursive-wedge-decomposition algorithm takes as input a map of polygonal terrain features. The recursive-wedge-decomposition implementation reported in [Ref. 21] was used in our research for initialization in our Chapter VI algorithms.

A generalization of the weighted-region problem allows anisotropic costs in regions, that is, costs which are a function of the direction of travel of the mobile agent, for example, in steeply sloped terrain. Ross [Ref. 40] solves the anisotropic weighted-region problem using a variation of recursive-wedge decomposition. Based on the effects of gravity, friction, and maximum force which can be applied by the agent, there are several sets of impermissible headings which may constrain travel across a polygonal region. A range of uphill headings may be ruled out by maximum force available, loss of traction, or catastrophic overturn, and a range

of sideslope headings may be ruled out by catastrophic overturn considerations. Additional optimality considerations include a range of downhill braking headings within which the agent must lose energy by braking, and Snell's Law for optimal paths as described above. Given these constraints, there are only four ways an optimal path can cross an anisotropic region. This insight leads to an algorithm which recursively decomposes groups of potentially optimal paths according to which terrain-feature vertices and edges they cross (*window sequences*), and applies A* search to these groups of paths, using various pruning criteria to limit the search space.

d. Optimal-Path Maps

Several researchers have used *optimal-path maps* (OPM), or as they are commonly called with respect to binary terrain, *shortest-path maps*, as a means of solving the shortest-path, binary-terrain problem. Lee and Preparata [Ref. 41] give an O(n log n) algorithm to construct an OPM for the special case that all obstacles are parallel line segments, and Reif and Storer [Ref. 42] give an O(mn + n log n) algorithm, where m is the number of obstacles and n is the number of obstacle vertices. Mitchell [Ref. 4] gives an $O(kn \log^2 n)$ algorithm for the general case, where k is an output-sensitive parameter somewhat related to the density of obstacles in the plane.

The algorithm of Lee and Preparata uses the plane sweep paradigm and constructs both the optimal-path tree and the planar partition with one sweep of the plane. Assuming without loss of generality that the parallel line-segment obstacles are vertical and the start point is to the left of all obstacles, the sweep line is also vertical and begins at the start point. The obstacles are indexed by their x-coordinates, and the initial event queue contains the x-coordinates of each obstacle. As the sweep line encounters an obstacle, it locates the two endpoints in regions of the OPM so far constructed and extends the optimal-path tree by inserting a node for each obstacle endpoint into the tree at the node associated with these regions. Then it constructs the three *bisectors*, or homogeneous-behavior region boundaries, which begin at the obstacle, two of which are rays and one of which is a hyperbola segment. It updates a list of "active" bisectors by deleting previously-found bisectors which intersect the current obstacle, and adds the new bisectors to the list. Then it updates the event queue by inserting points of intersection of the new bisectors with any other bisectors. Only the leftmost such intersection must be recorded. At each stage, the OPM is updated when both endpoints of a bisector are found. [Ref. 41]

The algorithm due to Reif and Storer takes as input a triangulation of the obstacle edges, and recursively processes these triangles to find shortest paths from the start point to each vertex of the triangulation. The algorithm "grows" outward from the start point, constructing a partition of the plane. The discussion of this algorithm in Reference 42 is somewhat obscure, as it does not use the terminology of shortest-path maps, and depends on other algorithms and data structures not fully explained in Reference 42.

A solution to the optimal-path-map problem which takes a different approach is presented by Payton [Ref. 43]. It is built on the wavefront propagation algorithm, and consists of storing the back-pointers for each cell. This array of pointers is called a *gradient field*, and provides information about which direction a mobile agent should go from any point on the map in order to travel along an optimal path. This approach could be used with other point-to-point path planners as well, although with greatly increased preprocessing time, by simply running the path planner for a finely-grained array of start points, and storing the initial direction of the resulting optimal path for each run.

Mitchell's algorithm introduces the concept of "generalized visibility" within the obstacle space, and constructs shortest-path maps for each new level of visibility. This algorithm begins by computing the visibility polygon from the start point, i.e., the polygon containing all points in the map which are not occluded from the start point by an obstacle edge. Then it appeals to the algorithm for constructing generalized Voronoi diagrams within simple polygons due to Aronov [Ref. 13] which takes into account that boundaries may be hyperbolic or linear, depending on the costs of optimal paths from obstacle vertices. Using this generalized-Voronoi-diagram concept, Mitchell's approach constructs a shortest-path map for the visibility polygon. Then, the algorithm computes the second level of visibility, that is, extends the visibility polygon to include all points visible from any part of the initial visibility polygon. Again, it reduces the problem of extending the shortest-path map to the problem of defining appropriate Voronoi-diagram problems on simple polygons. This process continues iteratively until all obstacles have been found by the generalized visibility process. [Ref. 4]

So essentially, Mitchell's algorithm is doing a concentric plane sweep (although not using this terminology), where at each iteration, the next generalized visibility polygon is found, a Voronoi diagram is constructed for the obstacles in the polygon, and these Voronoi diagrams are merged with the Voronoi diagram from the previous iterations. The computation of visibility polygons does use the plane sweep paradigm explicitly, sweeping a "geodesic" (or optimal) path angularly about the start point. In order to deal with several

cases in which a single sweep would not correctly identify all the event points, two sweeps, one in each direction about the start point, are done to compute each visibility polygon. This algorithm operates in $O(n \log^2 n)$ worst-case time, where n is the number of obstacle vertices in the input map. [Ref. 4]

The focus of this dissertation is on the construction of a planar partition for the weighted-region problem. In keeping with the convention discussed above of referring to solutions to the weighted-region problem as optimal paths instead of shortest paths, we refer to such a partition as an *optimal-path map*. Mitchell [Ref. 15], claims to have constructed an optimal-path map for the weighted-region problem, but does not mention the task of constructing region boundaries. His algorithm appears to construct, instead, an optimal-path tree, a necessary and time-consuming first step in constructing an optimal-path map, but gives little attention to construction of the planar partition. This confusion may arise from the fact that in the binary-terrain domain, construction of region boundaries is straightforward, a fairly insignificant part of the total problem, while the added complexity of the weighted-region problem creates additional complexities in the characterization of boundaries and the construction of the optimal-path map. In binary terrain, the standard Voronoi-diagram methods which construct straight-line bisectors only need to be extended to construct hyperbola segments as well, while in weighted-region terrain, such bisectors take on many different forms. Thus the problem of "defining the appropriate Voronoi-diagram problem", as Mitchell does in the binary case, is a much more difficult one.

III. MODIFYING WAVEFRONT PROPAGATION TO FIND SUB-OPTIMAL SOLUTIONS TO THE OPTIMAL-PATH-MAP PROBLEM

A. OVERVIEW

Wavefront propagation is well-suited as a method for solving the fixed-goal optimal-path-map problem (see Chapter II, Section C for a complete description of this problem), if the inherent error is acceptable in the application domain. The basic wavefront propagation algorithm can easily be extended by considering, for each cell on the wavefront, whether there should be a boundary between it and its adjacent cells, using one of the three definitions of "similar behavior". What for the point-to-point problem was a disadvantage of wavefront propagation, that the algorithm in its basic form searched blindly in all directions without regard to the location of the goal, becomes an advantage for the optimal-path-map problem because the paths from each cell in the map are available as a by-product of the algorithm simply by tracing the back pointers. Another advantage is that the asymptotic worst-case time complexity of the extension is the same as the basic algorithm.

In chapter II the path-generalizing function was defined in terms of "similar behavior" of paths. In this chapter we solidify the meaning of "similar behavior" to group paths in three different ways that make sense for wavefront propagation, thus defining the path-generalizing function in three ways. The first way produces boundaries between adjacent cells whose goal paths turn at cells which are not "equivalent". The second way uses a set of heuristics to group cells whose goal paths converge. The third way groups cells according to whether their paths turn at the same terrain-feature vertices and edges.

It might be possible to bypass the need for an optimal-path map altogether by simply storing back pointers for every cell in the map (for example, in the work of Payton discussed in Chapter II [Ref. 43], such a database of pointers is called a *gradient field*). Given a start cell's coordinates, the path to the goal could be reconstructed by following the pointers back to the goal cell. There are two disadvantages to this method. First, the average-case time complexity to reconstruct a backpath is O(n), for an input map of size n by n. Second, the storage requirement for the optimal-path map is $O(n^2)$. To avoid these problems, we store an optimal-path map.

B. MODIFYING THE PATH-GENERALIZING FUNCTION FOR WAVEFRONT PROPAGATION OPTIMAL-PATH-MAP CONSTRUCTION

1. The Pure Version of Wavefront-Propagation Optimal-Path-Map Construction

The most natural description of a path is the list of all cells from start point to goal point. Requiring two such path lists to be identical in order to represent "similar behavior" would result in every cell in the map comprising its own homogeneous-behavior region. But it is unnecessary to include all cells in a path segment which lie in on the same straight line. So another definition of a path list is the list of cells at which the optimal path turns, or more precisely, the cells in the backpath of a start point for which each back-pointer of the cell is in a different direction than the back-pointer of the cell's parent.

This definition still induces many distinct regions. A modification is to specify that two turn-point cells on different backpaths are considered equivalent if one of them lies on the first leg of the optimal-path list which starts at the other turn-point. Thus, for example, the two cells (5,3) and (6,2) in Figure 7 would have optimal-path lists [(5,5),(7,7)] and [(6,6),(7,7)] respectively; cells (5,5) and (6,6) would be considered equivalent because the optimal-path list of cell (5,5) is [(7,7)] and (6,6) lies on the line between (5,5) and (7,7); so cells (5,3) and (6,2) lie in the same region.

We call the version of the wavefront propagation optimal-path-map algorithm which uses this definition of the path-generalizing function the *pure* version, since it is based on a simple definition of homogeneous-behavior regions. Changes to the basic wavefront propagation algorithm in Appendix B necessary to implement this are presented in Table 1 below (two pages). The key change is a check for boundaries between each cell on the wavefront and its four neighbors. This is accomplished in procedure **expand-cell** which is executed once for each cell on the current wavefront. Procedure **expand-cell** calls procedure **check-for-boundaries** which compares the path lists of each of the cell's neighbors with the expanding cell's path list, checking for "equivalency" as defined above. Whenever a new cell is added to the wavefront, its path list is set by procedure **set-optimal-path-list** which is called from within **orthogonal-expand**, **diagonal-expand**, and **overflow**. These procedures, although not shown here, are modified from the versions shown in Appendix B by adding a call to **set-optimal-path-list** after each new cell is added to the wavefront list or the overflow list. When the procedure **check-equivalent-paths** called by **check-for-boundaries** determines that two path lists are not

TABLE 1

WAVEFRONT-PROPAGATION OPM ALGORITHM

```
algorithm wavefront-propagation-opm                    (Algorithm III-1)
   input: Goal-Point                                   /* REVISED from algorithm B-1 */
   {                                                   /*  in Appendix B. */
   Wavefront := Goal-Point;
   Boundary-List := empty list;
   while (Wavefront not empty)                         /* Iteratively expand wavefront */
      expand-wavefront(Wavefront);                     /* until nothing remains on it. */
   }                                                   /* end of wavefront-propagation-opm */.

procedure expand-wavefront                             /* REVISED PROCEDURE */
   input: Wavefront
   {
   if (Wavefront is empty)                             /* Base case of the recursion. */
      {
      Cells-for-New-Wavefront := empty list;
      New-Wavefront := empty list;
      }
   else
      {
      Current-Cell := cell on Wavefront with min remaining cost;
      expand-cell(Current-Cell);
      Rest-of-Wavefront := Wavefront less Current-Cell;
      expand-wavefront(Rest-of-Wavefront);             /* recursive call to expand-wavefront */
      New-Wavefront := Cells-for-New-Wavefront         /* Note: Wavefront is recursively emptied */
         appended onto front of New-Wavefront;         /* out level by level and New-Wavefront */
      }                                                /* is built up as each level returns. */
   }                                                   /* end of expand-wavefront */

procedure expand-cell                                  /* REVISED PROCEDURE */
   input: Current Cell
   {                                                   /* initialize flag assuming that Current-Cell */
   Finished-With-Cell := TRUE;                         /* will not stay on Wavefront */
   check-for-boundaries(Current-Cell);                 /* ADDED TO THIS VERSION */
   Boundary-List := New-Boundary-List appended         /* ADDED TO THIS VERSION */
      to Boundary-List;
   Cells-for-New-Wavefront := empty list;
   for (New-Cell := North-, East-, South-, and West-Neighbor)
      orthogonal-expand(Current-Cell,New-Cell);
   for (New-Cell := Northeast-, Southeast-, Southwest-, and Northwest-Neighbor)
      diagonal-expand(Current-Cell,New-Cell);
   if not (Finished-With-Cell)                         /* keep Current-Cell on Wavefront */
      Cells-for-New-Wavefront := Current-Cell appended
         onto Cells-for-New-Wavefront;                 /* CHECK FOR GOAL DELETED */

   }                                                   /* end of expand-cell */
```

TABLE 1 (CONTINUED)

WAVEFRONT-PROPAGATION OPM ALGORITHM

```
procedure check-for-boundaries                    /* NEW PROCEDURE */
   input: Current-Cell
   {
   New-Boundary-List := empty list;
   for (Neighbor-Cell := each of
      Current-Cell's eight neighbors)
      if not (Parent-Pointer of Neighbor-Cell = nil)   /* if wavefront has reached neighbor, */
         {                                             /* a boundary check can be made. */
         OPL1 := OPL-Parent of Neighbor-Cell;
         OPL2 := OPL-Parent of Current-Cell;
         if not (check-equivalent-paths(OPL1,OPL2))    /* update new boundary list */
            New-Boundary-List := edge or corner
               connecting the two cells appended to New-Boundary-List;
         }
   }                                              /* end of check-for-boundaries */

procedure set-optimal-path-list                   /* NEW PROCEDURE */
   input: Cell
   {                                              /* NOTE: There are two parent-pointer fields in */
   if (Parent of Cell is on line segment between  /* the "Cell" array - "Parent" field is predecessor */
      Cell and OPL-Parent of Parent of Cell)      /* of Cell on the backpath: "OPL-Parent" is the */
      OPL-Parent of Cell := OPL-Parent            /* predecessor on the Optimal-Path List. */
         of Parent of Cell;
   else
      OPL-Parent of Cell := Parent of Cell;
   }                                              /* end of optimal-path-list */

procedure check-equivalent-paths                  /* NEW PROCEDURE */
   input: OPL1, the OPL-Parent of Neighbor-Cell
      and OPL2, the OPL-Parent of Current-Cell
   output: returns TRUE if paths are
      equivalent, FALSE otherwise.
   {
   if ((first cell of OPL1 = first cell of OPL2)  /* Paths are equivalent if first */
      or (first cell of OPL1 is on the line       /* pair of cells are equivalent */
         between first and second cells of OPL2)
      or (first cell of OPL2 is on the line
         between first and second cells of OPL1)
      return(TRUE);
   else return (FALSE);
   }                                              /* end of check-equivalent-paths */
```

equivalent according to the above definition, the edge which the two cells share is considered a boundary and is added to a list of boundaries.

Since each cell with non-infinite cost is on the wavefront once during the algorithm, we will in the end check each cell on the map. For two adjacent cells, if one cell is has been reached by the wavefront and the other has not yet been reached, the second cell's path list will not yet be determined, so a boundary check is not yet possible. But when the second cell is finally put on the wavefront, its path list is set and a check of its neighbors will consider the first cell. So it is guaranteed that all pairs of neighbors will be checked by the end of the algorithm, and all boundaries between cells will be detected.

Note that references to the start point have been deleted from the algorithm, since we are looking for paths to all start points. The initial center of the wavefront is called the goal point. Also, there is no possibility for the algorithm to fail because of an inability to find the start point. When no cells remain on the wavefront, the program is done. Then the list of boundaries will be transformed into the appropriate data structure, a doubly-connected edge list, and the path information will be transformed into an optimal-path tree.

The procedure **set-optimal-path-list** will be called by procedures **orthogonal-expand**, **diagonal-expand**, and **overflow** each time a new cell is appended onto the Cells-for-New-Wavefront or Overflow lists respectively.

Figure 14 (on two pages) shows the result of applying the pure definition of the path-generalizing function to wavefront propagation, with a map consisting of a single obstacle and a single high-cost area. The figure shows successive snapshots over time as the wavefront expands and the back-pointers are set. The wavefront expands from the goal point in the center, and back pointers show the optimal path from each start point to the goal point. Homogeneous-behavior boundaries are shown as dotted curves. (Several horizontal backpaths appear darker than the others only because of the resolution of the printer used.) Figure 14a shows the first three snapshots, and Figure 14b shows the fourth snapshot and a final frame with backpaths removed for clarity.

Several homogeneous-behavior boundaries in Figure 14 are spurious, that is, are not predicted by theoretical analysis. (This analysis is presented in Chapter V.) Near the upper left corner of the high-cost area, for example, (see Frame 5 in Figure 14b) three straight homogeneous-behavior boundaries emanate from a point on the edge of the high-cost area: one is vertical, one is at a 45° angle, and one is horizontal. The latter

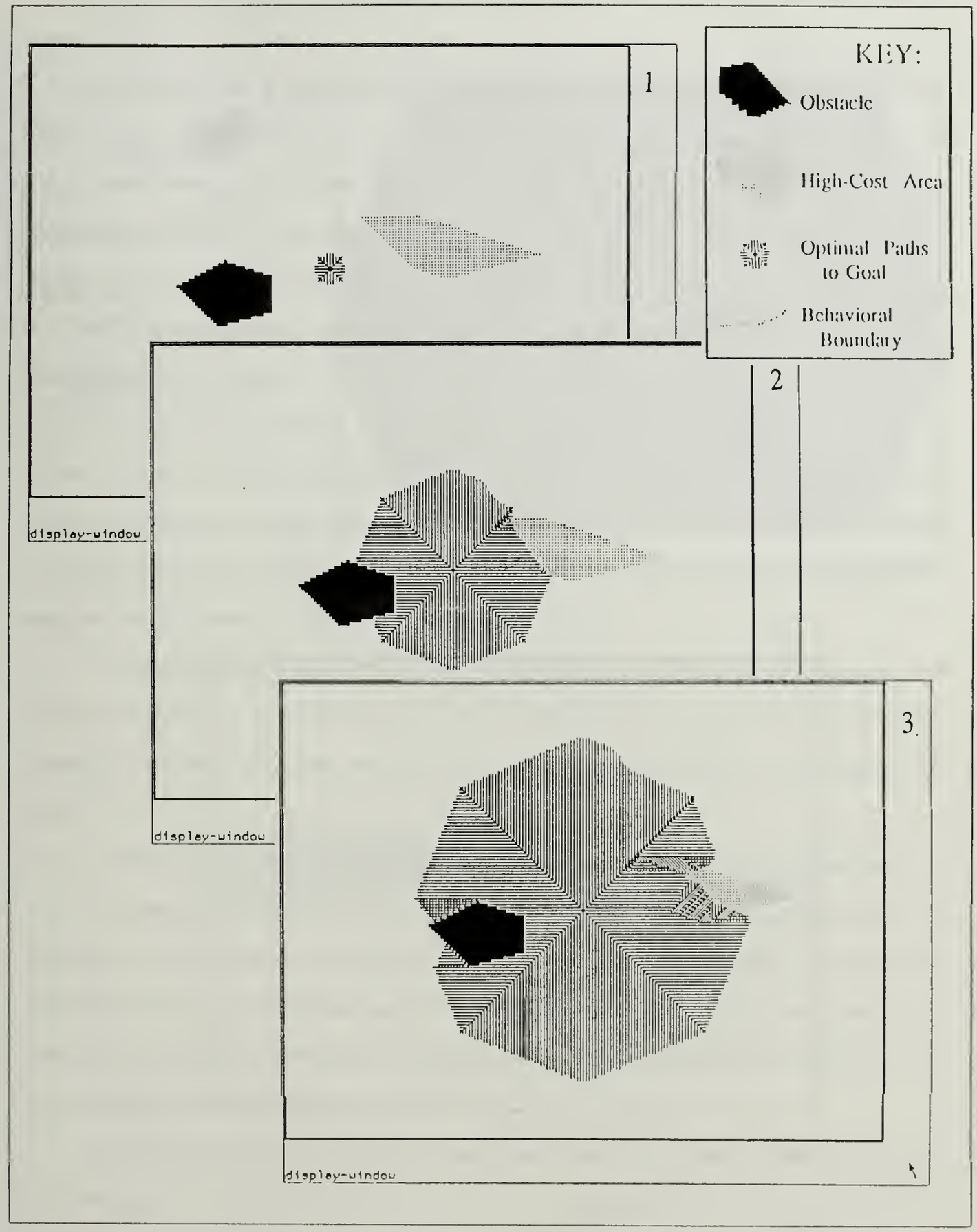


Figure 14a

Example of Pure OPM Version of Wavefront Propagation

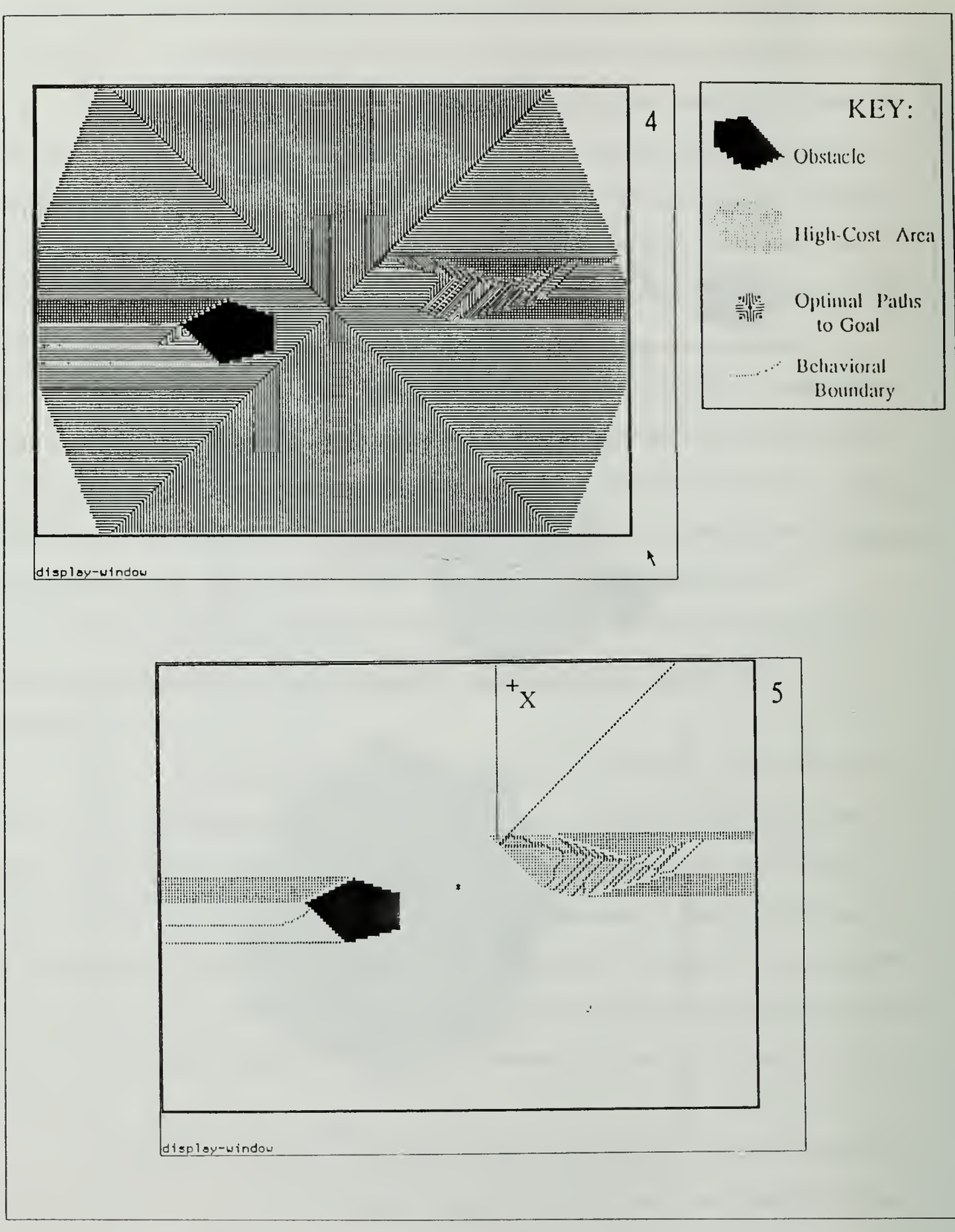


Figure 14b

Example of Pure OPM Version of Wavefront Propagation

two of these boundaries do not have analogues in the theoretical case, and the first, analogous to the "shadow" boundary expected at that vertex, is offset from the vertex of the high-cost area does not appear.

Some spurious boundaries are generated because straight lines are being approximated by piecewise-linear curves in the eight allowable propagation directions. Several examples occur to the right of the high-cost area and to the left of the obstacle. Multiple parallel boundaries are generated by the upper-left edge of the obstacle, although all but the topmost boundary are spurious, while the two lower boundaries generated by the lower-left edge are both predicted by the analysis of Chapter V. The reason for the difference is that the lower-left edge is positioned at a 45° angle to the vertical, allowing a single straight path to lie along it. Thus the above boundary-detection heuristic does not detect spurious boundaries along the edge because there are no turn points on the path. But the upper-left edge lies at less than a 45° angle with the horizontal, and so the path along it must "stair-step" its way to the upper vertex, causing boundaries to be generated. A similar error occurs along the upper right and lower right edges of the high-cost area, where the stair-step nature of the edges causes spurious exterior boundaries to appear. Further spurious boundaries occur in the inside of the high-cost area, and outside it just above its rightmost vertex.

Optimal-path maps generated by the pure wavefront propagation OPM algorithm will be useful if these spurious homogeneous-behavior boundaries do not matter. But there are approximately twice as many boundaries as are predicted by theoretical analysis, so storage and run-time speed are correspondingly less efficient.

2. The Diverging-Path Version of Wavefront-Propagation Optimal-Path-Map Construction

Another approach is based on the idea that two adjacent cells whose paths diverge should be in different regions, and so a boundary must exist between them. A way of detecting divergence of paths is to check the distance between the nth-generation ancestors of two adjacent cells. If the ancestors more than one cell apart, the cells are defined as diverging paths. In other words, we define the path-generalizing function so that it maps cells to sets of paths which do not diverge.

What should the value of n be? In other words, how far back along the paths of the two cells being compared should we check? If n is small, there will be fewer checks to perform, enhancing efficiency. If n is large some small terrain features may be overlooked by the divergence rule. On the other hand if n is large, we can handle situations, such as boundary emanating from the obstacle in Figure 14b, Frame 5, where back-

paths may parallel each other for some distance before diverging. But this situation can be taken care of by adding a second condition which says that two cells are in different regions if their parents are in different regions. Even with this rule, however, if n = 1, there are situations where the parents of two cells with diverging paths are adjacent; choosing n = 2 seems to give the best results. An additional necessary heuristic is that two paths are in different regions if a cell between the two ancestors being checked is a terrain feature cell. This handles special cases such as very acute obstacle vertices, or paths on opposite sides of a river. Figure 15 (on two pages) shows the result of applying these heuristics to wavefront propagation. We call this the *diverging-path* version of the wavefront-propagation OPM-generation algorithm.

So there are three heuristics used in the diverging-path version. First, adjacent cells whose second-generation ancestors are more than one cell apart are in different regions. Second, adjacent cells are in different regions if their parents are in different regions. Third, cells are in different regions if their second-generation ancestors have a terrain-feature cell between them.

This variant algorithm is not much better than the pure variant, as can be seen by studying Figure 15b, Frame 5. Here too few boundaries are generated, and a few spurious boundaries appear as well. Those boundaries defined in Chapter V as *opposite-edge boundaries*, i.e., boundaries which distinguish between paths which go in opposite directions around a terrain feature, are the ones best detected by the diverging-path version. *Shadow boundaries*, i.e., boundaries which distinguish between paths which go through a terrain-feature vertex from those which bypass it, are not detected at all. Spurious boundaries arise within homogeneous-cost areas. The homogeneous-cost area in Figure 15 has spurious boundaries just above its rightmost vertex. But for purely binary terrain, i.e., obstacles on a homogeneous-cost background, the diverging-path version may be appropriate.

3. The Vertex-Edge Version of Wavefront-Propagation Optimal-Path-Map Construction

Any variant algorithm that relies solely on the turns in a path will misinterpret some turns as due to the terrain when in fact they were due only to the mechanics of the algorithm (e.g., the eight propagation directions), and vice versa. Also, the diverging-path variant only detects a certain class of boundary. A way to attack both of these problems is to plot boundaries based on how terrain-features affect optimal paths.

In terrain with piecewise-linear edges in homogeneous-cost background, optimal paths will turn only at terrain feature vertices or edges (Theorem I-2, Appendix A). Thus, if a turn in a path occurs at other than a

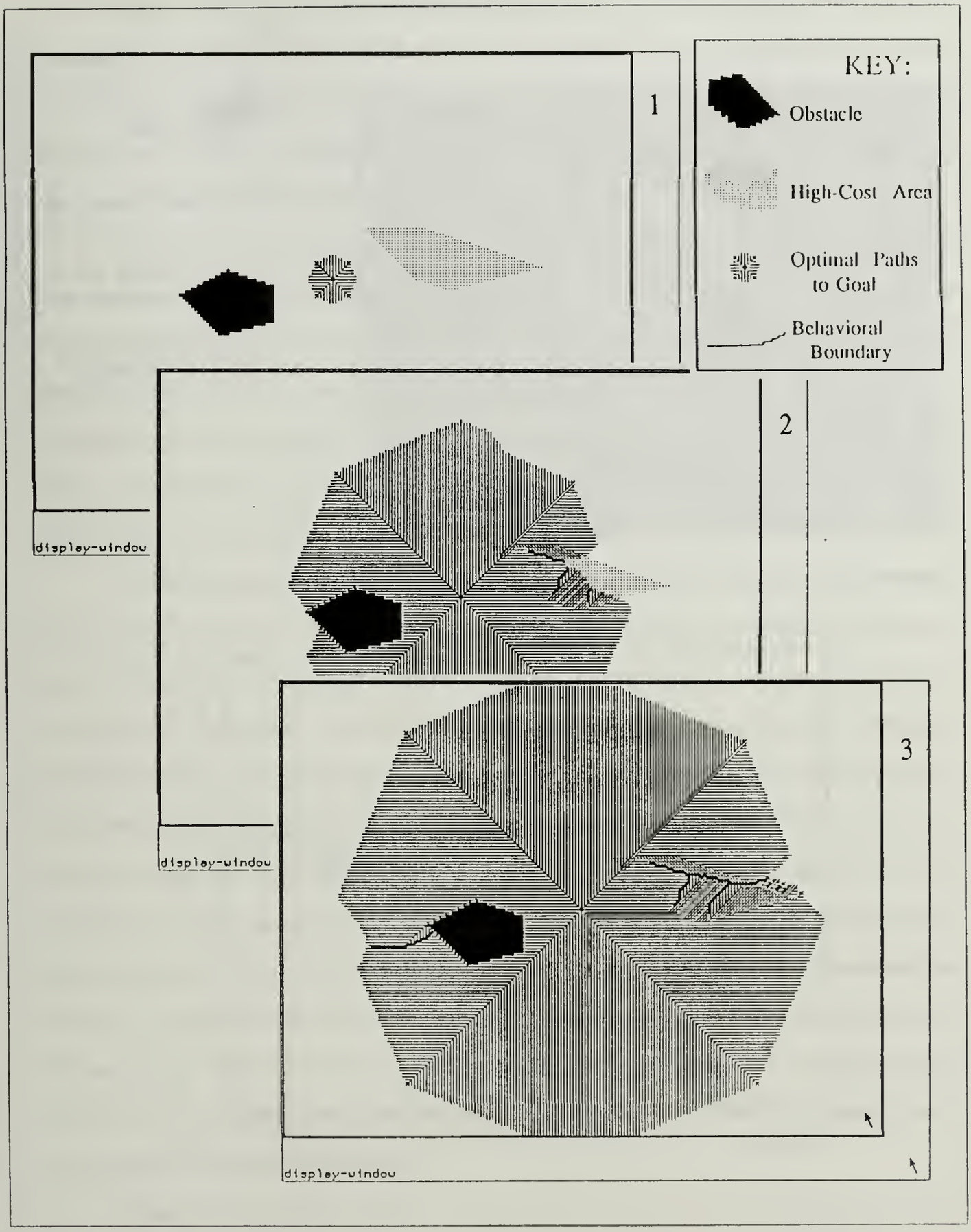


Figure 15a

Example of Diverging-Path OPM Version of Wavefront Propagation

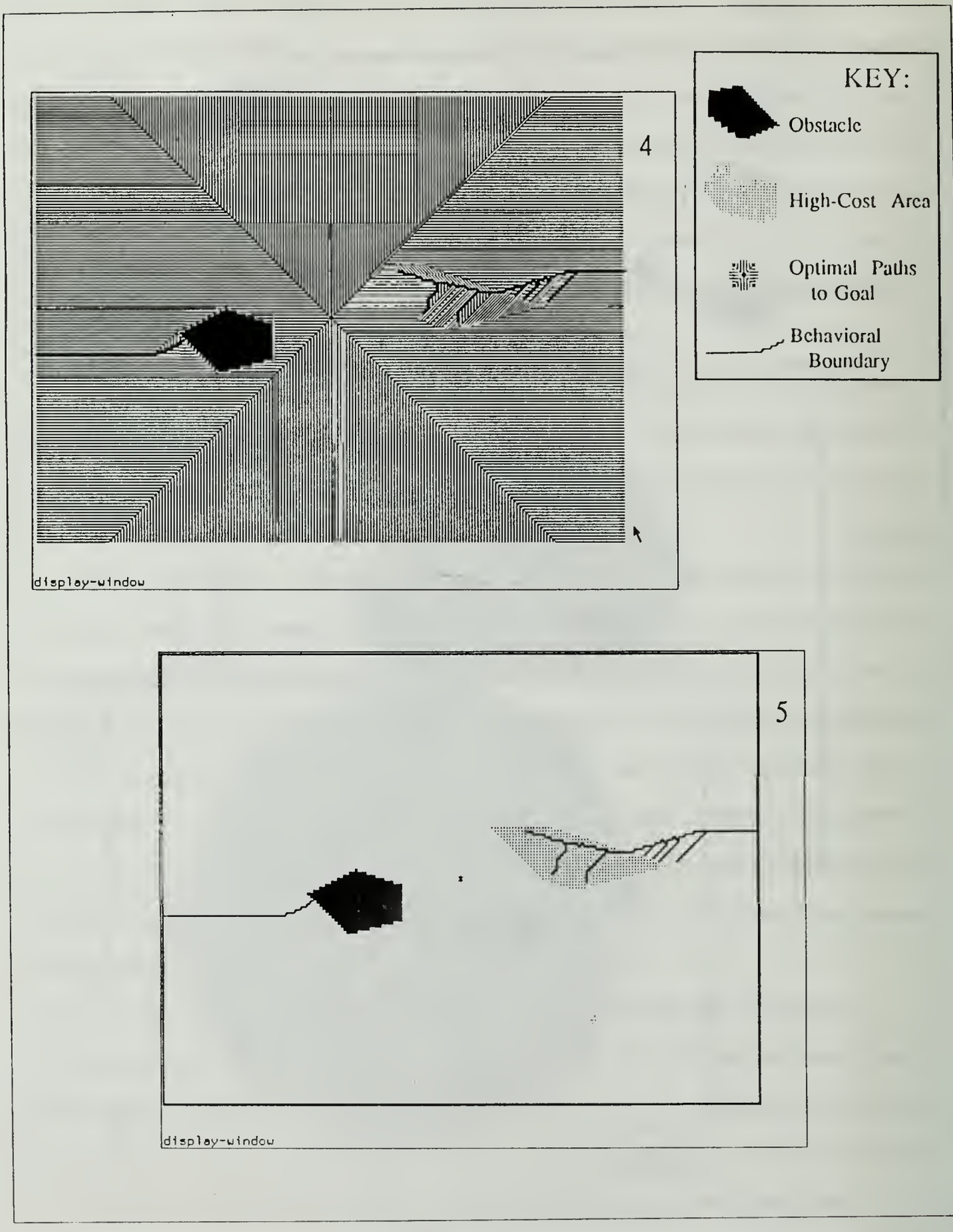


Figure 15b

Example of Diverging-Path OPM Version of Wavefront Propagation

vertex or edge, it must be turning based on algorithm mechanics alone. So we could check whether or not a turn point in a path is adjacent to a terrain-feature vertex or occurs at the edge of a homogeneous-cost region. We could define the path-generalizing function as mapping a cell to a list of the terrain-feature vertices and edges at which its optimal path turns. We can say that a path turns *at* a vertex or edge if the turn cell is adjacent to or the same as the vertex or edge cell.

This approach requires some additional terrain preprocessing. Since terrain in the two previous versions has been represented entirely as individual cells, some way of finding and representing terrain-feature edges and vertices will become necessary. Such a preprocessing algorithm could group cells into terrain features of homogeneous cost, and then fit polygons to each feature. For each vertex of the polygon, it could find the closest corresponding cell in the original representation and label it as a vertex. For each edge of the polygon, it could find which cells most closely corresponded to it and label them as lying on that edge. Wade [Ref. 44] presents an algorithm for doing such terrain preprocessing.

So we redefine "path list" to include only vertex and edge descriptors. To do this, turn cells are checked to see if they are adjacent to a terrain-feature vertex or edge. This procedure may create a spurious boundary if a path turns twice within one cell of a vertex, a case which would happen at a corner which formed a very acute angle, for example, a river end. In this case, a spurious boundary would lie along the side of the river segment away from the start point. We must also specify from which side a path crosses an edge, because a path may leave an area across an edge and then reenter it across the same edge. This type of path is illustrated in Figure 16b, Frame 4, starting at the cell labeled A. The path from A has a path list [A,C,D,G], while a path from cell B has a path list [C,D,G]. When comparing cells A and C (the first cells on the two paths) to determine if A and B have a boundary between them, we must be able to determine that the first path crosses *out of* the high-cost area at A, while the second path crosses *into* the area at C, and so the paths have different behavior. This set of heuristics provides the ability to detect boundaries inside homogeneous-cost areas, across rivers, and across roads. The procedures **set-optimal-path-list** and **check-equivalent-paths** are listed in Table 2 with the appropriate changes.

Figure 16 shows the above heuristics in operation. There is a very close correspondence between the boundaries of Figure 16b, Frame 5, and the theoretically correct boundaries for an exact optimal-path map. Few spurious boundaries are generated. For example, there are too many boundaries emanating from the far

TABLE 2

WAVEFRONT-PROPAGATION OPM ALGORITHM

CHANGES FOR VERTEX-EDGE VERSION

```
procedure set-optimal-path-list                    /* REVISED PROCEDURE */
   input: Cell
   {
   if (Parent of Cell is on Edge_i and OPL-Parent of   /* include a cell in OPL for each boundary- */
      Parent of Cell is not on Edge_i)                 /* crossing episode. */
      OPL-Parent of Cell := Parent of Cell;
   else if (Parent of Cell is on line segment between  /* SAME AS PREVIOUS VERSION */
      Cell and OPL-Parent of Parent of Cell)
      OPL-Parent of Cell := OPL-Parent of Parent of Cell;
   else
      OPL-Parent of Cell := Parent of Cell;
   }                                                   /* end of optimal-path-list */

procedure check-equivalent-paths                   /* REVISED PROCEDURE */
   input: OPL1, the OPL-Parent of Neighbor-Cell
      and OPL2, the OPL-Parent of Current-Cell
   output: returns TRUE if paths are equivalent, FALSE otherwise.
   {
   if (OPL1 = OPL2 = [goal-point])
      return(TRUE);
   else
      {
      for i = 1 to 2
         until ((first cell of OPL_i is adjacent to
            a cell marked "vertex") or (first cell of   /* Consider only cells which are */
            OPL_i is marked "edge_j"))                  /* adjacent to terrain-feature vertices */
            OPL_i := OPL_i less first cell;             /* or represent edge-crossing episodes */
      if ((first cell of OPL1 = first cell of OPL2)     /* Paths are equivalent if each */
         or (first cell of OPL1 is on the line          /* pair of cells are equivalent */
            between first and second cells of OPL2)
         or (first cell of OPL2 is on the line
            between first and second cells of OPL1)
         or ((edge_i = edge_j) and                      /* NEW CONDITION */
            check-equivalent-paths(OPL1 less first
            cell, OPL2 less first cell))
         Boundary-Flag := TRUE;
      else Boundary-Flag := FALSE;
      }
   }                                                   /* end of check-equivalent-paths */
```

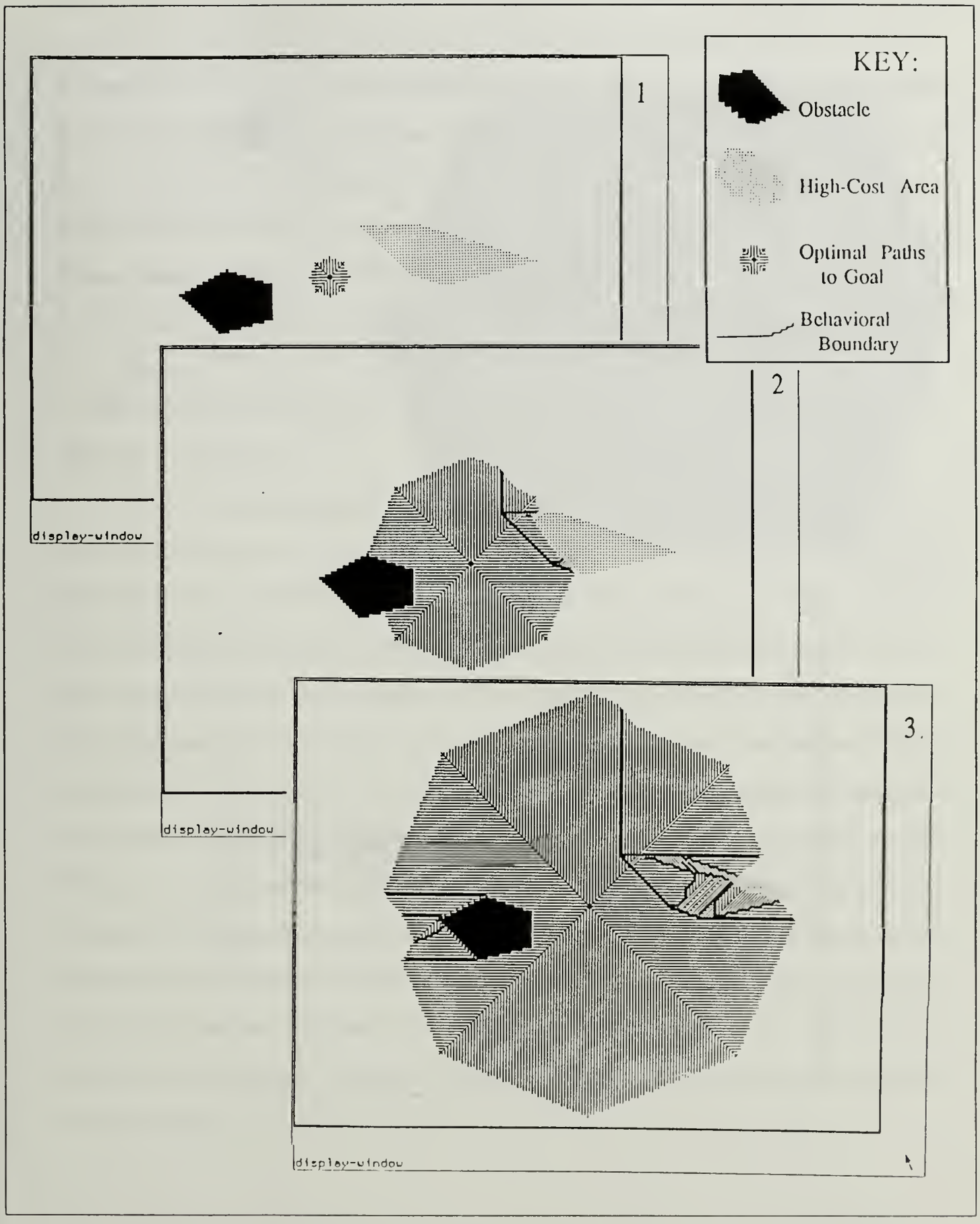


Figure 16a

Example of Vertex-Edge OPM Version of Wavefront Propagation

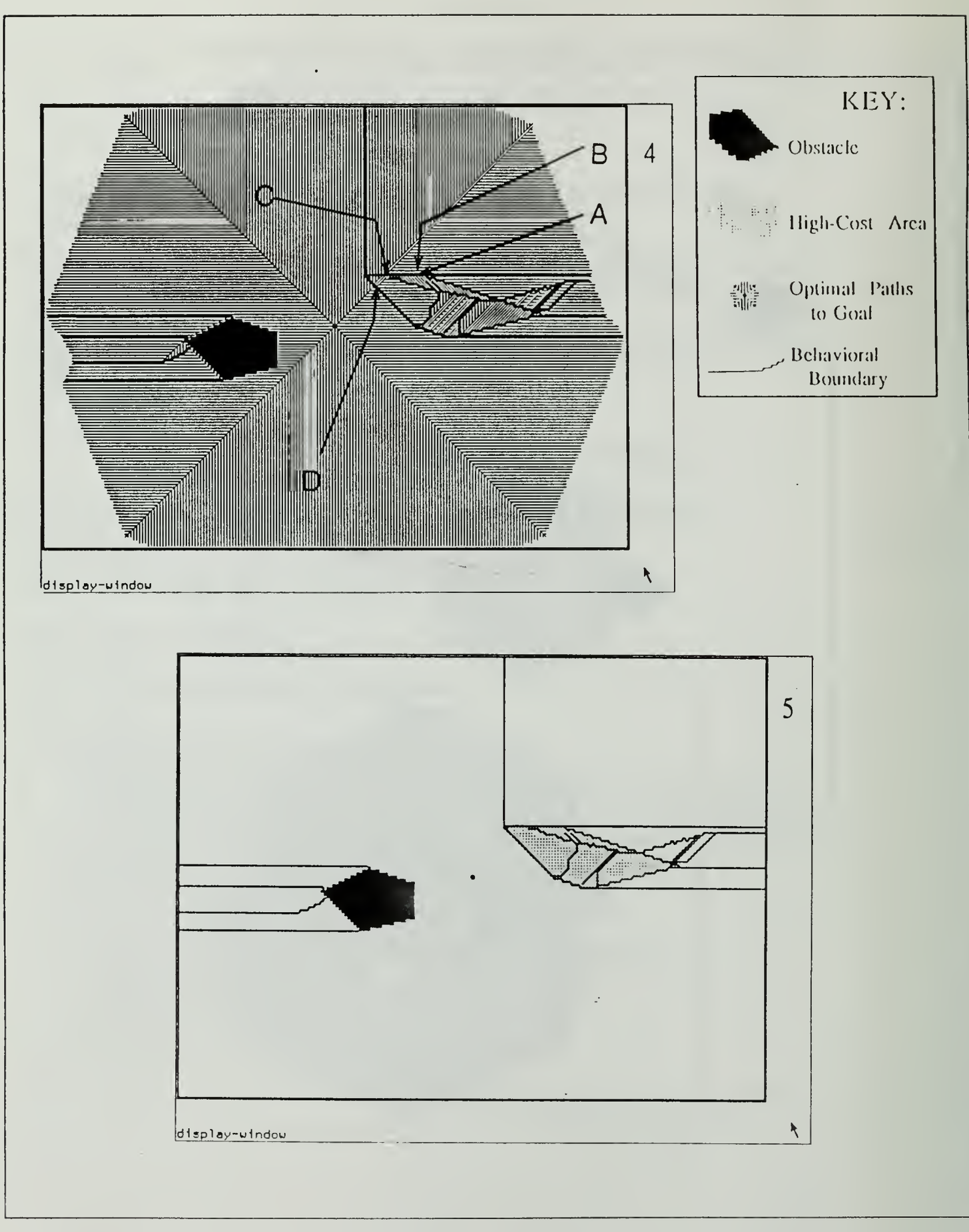


Figure 16b

Example of Vertex-Edge OPM Version of Wavefront Propagation

right vertex of the high-cost area. And few boundaries are overlooked, although some shadow boundaries do not appear. For example, a shadow boundary should emanate from the lower right vertex of the obstacle; in Chapter V we develop analytic characterizations of homogeneous-behavior boundaries and find that the linear boundaries incident upon vertices should act as if they were shadows with the goal acting as a point light source. Also, the curved boundaries on the hidden side of obstacles should be hyperbolas, and the curved boundaries inside homogeneous-cost areas should have monotonic curvature. From these comments, it can be seen that the boundaries generated by the vertex-edge version (as well as the other versions of wavefront propagation) have some error in location and shape, although they may suffice for many applications.

C. RECONSTRUCTING OPTIMAL PATHS FROM WAVEFRONT PROPAGATION OPTIMAL-PATH MAPS

How can we reconstruct the optimal path from the start point knowing the node of the optimal-path tree which descibes its behavior? The answer depends on what information is available in the nodes, which will be different depending on the version of wavefront propagation, because homogeneous-behavior regions are defined differently for each version. For the pure or diverging-path version, each optimal-path-tree node represents a single cell. Because intermediate turn cells on the portion of a path which lies within a homogeneous-behavior region are a result of the mechanics of the algorithm, and not of terrain-feature influence, a path can be approximated by plotting straight lines from a start cell to the cell of the node representing the region in which the start cell lies, another straight line from that cell to its parent in the optimal-path tree, and so on back to the goal. This type of path no longer conforms to the grid-based model; otherwise, some "stair-step" approximation of the line would be required. By Theorem I-2, in the type of terrain considered herein, paths are straight-line segments except at terrain-feature vertices or edges. So the vertex-edge version can use the above method for paths from start cells to nodes representing vertices, and between vertices. Between edges, further processing would be necessary to determine where along an edge a given path would cross using Snell's Law as discussed in Chapter II.

IV. ANALYSIS OF WAVEFRONT-PROPAGATION OPM-GENERATION ALGORITHMS

A. SOURCES OF ERROR IN WAVEFRONT-PROPAGATION OPM-GENERATION ALGORITHMS

A problem with using wavefront propagation to generate optimal-path maps is that the inherent error of the algorithm is carried forward to the OPM. As stated previously, Richbourg [Ref. 20] showed that an upper bound on the error factor of the cost of a model-optimal path generated by the point-to-point wavefront propagation algorithm compared with the cost of the corresponding real-world optimal path is $\cos(\pi/8)$, or about 7.6%. The fact that the shapes of boundaries generated by wavefront propagation are only approximations of the correct shapes derived in Chapter V reflects the error in the shape and cost of optimal paths inherent in wavefront propagation.

The optimal-path map in our approach does not retain information about all the intermediate cells where each path turns, and so we cannot reproduce the path exactly as generated by wavefront propagation. If we could, however, the upper bound on percent error of 7.6% would remain in effect, because nothing in the OPM algorithms of Chapter III affected how the wavefront expanded from cell to cell. The backpaths of Figures 14, 15, and 16 are all identical (compare Frame 4 of each figure), and only the boundaries differ. Although we cannot reconstruct a path exactly as generated by wavefront propagation, the straight-line approximation method proposed in Section C of Chapter III actually produces a path as good or better in cost than the wavefront propagation path. Straight-line approximations of a path always go through the region root, which was on the original path. They also lie completely within an area of homogeneous cost, because homogeneous-behavior regions are star-shaped with respect to the region root (Corollary I-1.4). By the triangle inequality, their cost is always less than or equal to the original path. So since costs of straight-line approximations are lower bounds on costs of wavefront-generated paths, the previously stated upper bound on percent error remains a valid upper bound. Can this upper bound be improved?

Although in the case of most start cells, substantial improvement over the cost of model-optimal paths generated by wavefront propagation will be achieved by this path reconstruction method, the upper bound on

error cannot be tightened in general, because there will be situations where the error in placement of a boundary would cause a start point to be placed in an incorrect region, (although without exceeding the upper bound). An example of such a case occurs in Frame 5 of Figure 16, at the point labeled X. The optimal path from point X should be a straight line to the goal. But since wavefront propagation caused error in the placement of the vertical boundary (it should have been a "shadow boundary," a ray from the upper left vertex of the high-cost area extending directly away from the goal point), X is to the right of the vertical boundary instead of to its left, so it is associated with the region whose root is the top-left edge of the high-cost area. Thus, a reconstructed path will go in a straight line to the top of the high-cost area, and then cut across its corner and go the the goal. This path has a cost error close to the original upper bound.

Thus the upper bound on percent error of the cost of wavefront-propagation-generated model-optimal paths with respect to real-world optimal paths remains as stated for the point-to-point version of the algorithm, i.e., 7.6%, although average error will be improved by appropriate reconstruction of paths from the optimal-path map.

B. TIME COMPLEXITY OF WAVEFRONT-PROPAGATION OPM-GENERATION ALGORITHMS

As stated in Chapter II, point-to-point wavefront propagation implemented using Dijkstra's algorithm has worst-case time complexity O(m log m), where there are m cells in the input map. In Algorithms B-1 (Appendix B) and III-1 (Chapter III), however, the algorithm is modelled on the wavefront analogy, and Dijkstra's algorithm is not followed exactly (because cells may remain on the wavefront for more than one iteration, and a search for the minimum-cost edge is not done for each wavefront). As explained in Chapter II, the time complexity of this version is O(c m), where c is the maximum cost of a cell in the input map, time is incremented by 1 unit each step, and it is assumed that there is some upper bound on the size of c.

The mechanism for detecting boundaries is to check each cell on the wavefront against each of its eight neighbors. There are eight, or O(constant) checks for each of the m cells in the map. Each boundary check in the *pure* version consists of an O(constant) comparison of the first turn points on the backpaths of the two cells being checked. So boundary-checking takes O(m) time. This is added to the time for the basic algorithm, so

the *pure* version of wavefront-propagation optimal-path-map generation has the same asymptotic worst-case time complexity as point-to-point wavefront propagation, or O(c m).

In the *diverging-path* version of wavefront-propagation optimal-path-map generation, the boundary check consists of a comparison of the distance between the parents of the parents of the two cells. Again it is an O(constant) operation to follow two back-pointers for each cell and compute a distance, so boundary-checking takes O(m) time, and the *diverging-path* version is also O(c m).

The *vertex-edge* version of wavefront-propagation optimal-path-map generation uses the same path lists as the *pure* version, but considers only so-called *distinguished cells* on the lists. A check of the first two distinguished cells in a path list by procedure **check-equivalent-paths** will give a conclusive answer about whether or not two paths are "similarly behaved". This check is an O(constant) process where the first element in each list is retrieved, and the two elements compared. So the vertex-edge version also has worst-case time complexity O(c m).

As discussed in Chapter III, the vertex-edge version requires preprocessing of the terrain to fit groups of homogeneous-cost cells to polygons or line segments, and to find vertices and edges. The algorithm of Wade and Rowe [Ref. 44] which does this has two passes. The first pass processes each cell once, in total O(m) time. The second pass is recursive, and a worst-case time complexity is not given, but for a map with k edge cells, is approximately O(log k). Under the above assumptions, the number of edge cells is significantly less than the number of cells, so k<m. Therefore the terrain preprocessing is dominated by the wavefront propagation algorithm.

C. SPACE COMPLEXITY OF WAVEFRONT-PROPAGATION OPM-GENERATION ALGORITHMS

The space required for the point-to-point wavefront propagation algorithm is simply O(m), where the input map has m cells. Storage is usually implemented by a $\sqrt{m}$ by $\sqrt{m}$ array which holds cost information and a pointer to the parent of the cell on its backpath. During execution, another data structure will hold the coordinates of those cells currently on the wavefront. When the algorithm is expanded to deal with the two-dimensional, or optimal-path-map case, several new data structures must be added. First, for the pure and the vertex-edge versions, two fields must be added to the cell array to hold the coordinates of the cell's parent on

the optimal-path list (in general not the cell's parent on the backpath). Secondly, new data structures must be added to hold the output. These data structures are the DCEL and the optimal-path tree.

As explained in Chapter II, Section B, a doubly-connected edge list (DCEL) along with an optimal-path tree are well suited to representing the optimal-path map. The size of the optimal-path tree is proportional to the number of homogeneous-behavior regions in the optimal-path map, since there is one node per region. Since in the worst case there could be no more than one region per cell, the optimal-path tree will never require more than O(m) storage. In fact as discussed above, the number of regions is assumed to be significantly larger than the number of cells, so the optimal-path tree will only require a small fraction of the total number of cells in the input map, and is more accurately a function of the number of terrain feature vertices and edges, or $O(v + e)$.

The DCEL represents the planar partition by listing characteristics of each line segment, or edge, in the partition. Since each segment of the wavefront-propagation-OPM boundaries is designated as lying between two specified cells, there can never be more than O(m) boundary segments, and in fact, the one-dimensional nature of boundaries will tend to produce an DCEL of $O(\sqrt{m})$ size. In terms of terrain-feature vertices and edges, it is shown in Chapter V that any given vertex or edge has a constant number of region boundaries associated with it, so the DCEL will have size of $O(v + e)$. Note that the O(m) input map can be discarded after preprocessing, so the amount of storage needed at run-time will be $O(v + e)$.

In practice, a great amount of storage can be saved in the way the planar partition is represented by the DCEL. As produced by the wavefront-propagation OPM-generation algorithm, boundaries are represented by lists of cell edges (perhaps implemented simply by listing coordinates in the same coordinate system as the cells, but incremented or decremented by .5). But in fact, boundaries in the grid-based domain typically contain long, near-linear sequences, so the number of edges in the DCEL can be reduced greatly by representing only endpoints of such sequences. Figure 16 shows about half of the boundaries to be linear.

D. EMPIRICAL PERFORMANCE OF WAVEFRONT-PROPAGATION OPM IMPLEMENTATIONS

The three versions of the OPM-generation algorithm described in Section B of Chapter III were implemented in Common Lisp on a Symbolics 3620 Lisp Machine. Although no special effort was made to make

these implementations efficient, some idea of the relative performance of the four versions, and some rough idea of the performance of wavefront propagation in general, can be gained by observing actual run-times. Table 3 shows average elapsed times for two typical input maps, based on the Lisp function "get-universal-time". These real-time figures give some rough idea of the actual performance of these implementations.

TABLE 3

WAVEFRONT-PROPAGATION OPM-GENERATION

RELATIVE PERFORMANCE OF THREE VERSIONS

Map Number	Pure OPM Version	Diverging-Path Version	Vertex-Edge Version
1			
(average CPU Time)	449,759 cycles	793,094 cycles	2,292,827 cycles
(average Real Time)	493 sec	843 sec	2,440 sec
2			
(average CPU Time)	1,558,722 cycles	916,535 cycles	2,013,910 cycles
(average Real Time)	1,714 sec	973 sec	2113 sec

NOTES:

(1) Average CPU Time is elapsed time as per machine-dependent LISP function "get-internal-run-time" averaged over eight runs.

(2) Average Real Time is elapsed time as per LISP function "get-universal-time" averaged over eight runs.

(3) Versions were implemented in Common-Lisp on a Symbolics™ 3640 operating under Genera 4.1™.

(4) Map 1 was 199 by 150 cells (i.e., 29850), with one obstacle and one high-cost feature, 12 vertices and 12 edges, with 465 cells, or 1.5%, of infinite cost (obstacle cells) and 741 cells, or 2.5%, of cost two.

(5) Map 2 was 199 by 150 cells (i.e., 29850), with three obstacles, 15 vertices and 15 edges, with 619 cells, or 2.1%, of infinite cost.

V. CHARACTERIZATION OF REGION BOUNDARIES

In this chapter, we formulate the geometrical groundwork necessary for an OPM construction algorithm which relies on spatial reasoning to eliminate much of the inaccuracy inherent in the wavefront propagation OPM construction algorithm. The algorithm applies to maps consisting of the five types of terrain defined in Chapter I, Section E, obstacles, roads, rivers, homogeneous-cost areas (HCA), and homogeneous-cost background. The approach we use is to determine the analytic characteristics of boundaries between regions of similarly-behaved optimal paths as functions of terrain feature characteristics. It turns out that all boundaries associated with the first three of the above terrain feature types (roads, rivers, and obstacles) are segments of conic sections. Boundaries associated with HCA's are more mathematically complex, and in many cases do not appear to have closed-form expressions. In addition to the algebraic form of these boundaries, we develop the theory which describes the circumstances in which each type of boundary occurs. The algorithms described in Chapter VI will rely on the results developed in this chapter for the basic steps involving construction of each boundary.

First, *primitive* terrain features, that is single polygonal obstacles and homogeneous-cost areas, and single river and road line segments, are studied and the boundaries they generate are characterized. Then a unifying theory is introduced which underlies all types of boundaries as they occur in terrain as defined herein. Development of algorithms for constructing OPM's for each of the primitive terrain features and for combined terrain is deferred until Chapter VI. Appendix C contains additional examples of optimal-path maps for each of the primitive terrain features presented.

A. REGION BOUNDARIES ASSOCIATED WITH PRIMITIVE TERRAIN FEATURES

Table 4 summarizes the types of homogeneous-behavior-region boundaries associated with each type of primitive terrain feature. Each type of terrain feature is listed in the left column. The second, third, fourth, and fifth columns contain the names of the boundary types associated with that terrain feature which are linear, parabolic, hyperbolic, and non-conic respectively. Since there are four cases of homogeneous-cost area (HCA) depending on whether the goal is inside or outside the HCA and on whether the HCA cost is higher or lower than the surrounding terrain, each of which has distinctively different boundaries, these four cases are listed

TABLE 4

SUMMARY OF HOMOGENEOUS-BEHAVIOR-REGION BOUNDARIES BY TERRAIN TYPE

	FORM OF BOUNDARY			
	LINEAR	PARABOLIC	HYPERBOLIC	NON-CONIC
Obstacle	Shadow(c/c) Obstacle-edge		Opposite-edge(c/c)	
River Segment	Shadow(c/c) River-edge		River-opposite-edge(c/c) River-crossing(c/c)	
Road Segment	Road-edge Rd-end/road-tvlg(c/p) Rd-tvlg/road-crossing(p/c) Shadow(c/c)	Near-side-road-travelling/goal(p/c) Far-side-road-travelling/goal(p/c)	Road-end/goal(c/c)	
High-Cost Exterior-Goal HCA	HCA-edge Hidden-edge/merging-path(p/p) Hidden-edge/diverging-path(p/p) Shadow(c/c)		Opp-edge-0-thru-interior(c/c) Opp-edge-1-thru-interior(c/d) Opp-edge-2-thru interior(d/d)	Visible-edge(d/d) Visible-hidden-edge(d/p) Corner-cutting(c/d)
High-Cost Interior-Goal HCA	HCA-edge Shadow Hidden-edge(p/p) Interior-opposite-edge(p/p)	Hidden-edge/goal(p/c) Visible-edge/goal(p/c)	Exterior-opposite-edge(c/c)	Corner-cutting (c/d) Visible-edge(d/d)
Low-Cost Exterior-Goal HCA	HCA-edge Vertex/edge-following(c/p) Vertex-edge-crossing(c/d)	Edge-following/goal(p/c)	Vertex/goal(c/c)	Edge-xing/(d/c) Opposite-edge(d/d) Visible-edge(d/d)
Low-Cost Interior-Goal HCA	HCA-edge Vertex/edge-crossing(c/d)			
Multiple-Connected River Segments	Shadow(c/c) River-edge		River-opposite-edge(c/c) River-crossing(c/c) Near-side-river-crossing(c/c)	

separately. Also listed with each boundary name is a coded description of what type of cost functions are associated with the homogeneous-behavior regions on either side of the boundary. The code "c" means the cost function of a region is conical, "p" means it is planar, and "d" means it is a "distorted cone". Terrain-feature edges always form boundaries, which of course are linear since terrain-feature edges are linear, but are not associated with a particular cost function, so no code is shown. (See Section C for a discussion of cost functions.)

1. Obstacles

We begin by characterizing boundaries associated with a single obstacle in homogeneous-cost background terrain (see Theorem V-1, Appendix A). (The types of boundaries associated with obstacles have previously been determined by Mitchell [Ref. 4] using different terminology.) With respect to obstacles, define a *visible edge* to be an edge for which no point on the edge has an optimal-path list whose first element lies on the obstacle perimeter. Define a *hidden edge* as a non-visible edge, i.e., an edge for which some point on the edge has an optimal-path list whose first element lies on the obstacle perimeter. In the case of terrain containing only a single obstacle, this means that both visible-edge vertices are visible to the goal point. In Figure 17, edges AB and BC are visible edges. Edges CD, DE, and EA are hidden edges. (Many of the following figures are similar in format. Terrain features are shown as polygons or line segments. Homogeneous-behavior-region boundaries are shown as solid curves. Occasionally continuations of the boundaries are shown as dashed lines to clarify the form of a boundary. In many of the figures a field of small vectors represents the initial direction of the optimal paths from a sampling of start points. These fields are not part of the optimal-path map, but serve to illustrate the directions paths take and to corroborate the correctness of plotted boundaries.) Define an *opposite edge* to be the obstacle hidden edge for which the optimal path lists of neither vertex includes the other. An isolated obstacle has exactly one opposite edge (Lemma V-1.3, Appendix A). Edge DE is the opposite edge in Figure 17. A special case is that in which the role of the opposite edge is assumed by an obstacle vertex; this is ruled out by the general position assumption discussed in Chapter I, although the analysis for including such a case is a simple extension of the below. Define an *opposite point* as the point on the opposite edge with two distinct optimal paths, one through each vertex of the opposite edge.

There are three types of boundaries associated with obstacles. *Obstacle edges* are trivial boundaries, since they separate regions whose optimal-path lists are [[], goal-point] from regions with non-degenerate optimal-path lists (see Lemma V-1.1). *Obstacle shadows* emanate from vertices of hidden edges in a straight

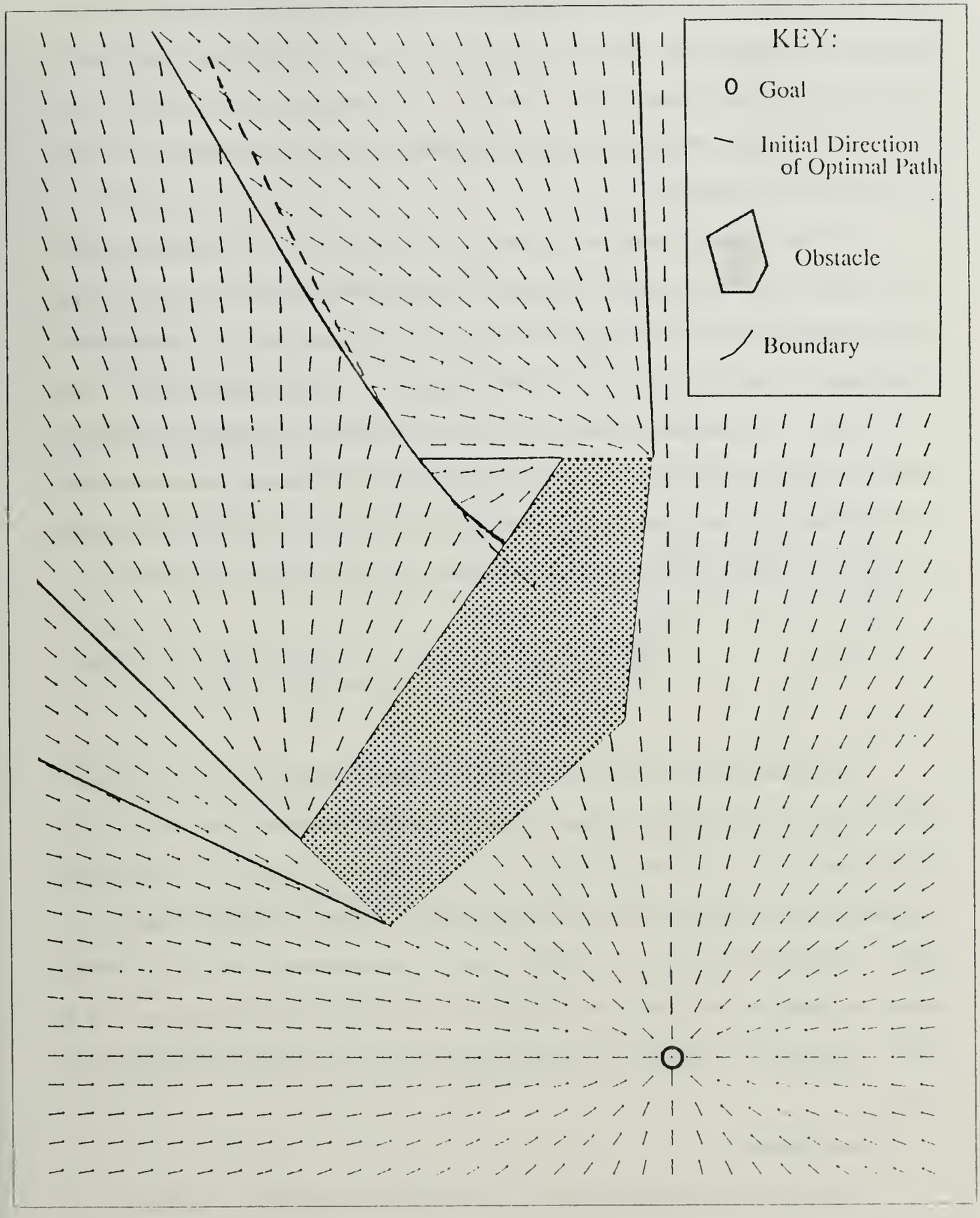


Figure 17

Obstacle

line, as if the goal were a point light source; each vertex of a hidden edge generates a shadow boundary. For those vertices which join a hidden edge and a visible edge, the line segment lies on the line defined by the vertex and the goal; for those vertices which join two hidden edges, the line segment lies on the line defined by that vertex and the vertex of the hidden edge which is included in the first vertex's optimal path. (See Figure 17 and Lemma V-1.2, Appendix A).

Each obstacle also has exactly one *opposite-edge boundary* which emanates from the opposite edge of an obstacle, and consists of segments of hyperbolas. This follows directly from the definition of a boundary by the application of basic analytical geometry (see Lemma V-1.4, Appendix A). The hyperbola is defined by considering the vertices V_1 and V_2 of the opposite edge as foci. Choosing a coordinate system such that the x-axis intersects both foci and the origin is mid-way between them, Equation 1 describes the opposite-edge boundary. Forcing constant a to be positive restricts Equation 1 to the one branch of the hyperbola which is closer to the higher-cost focus. The segment of this branch which is active as a boundary begins at the point on the opposite edge intersected by the branch and continues away from the obstacle. (See Figure 17).

(Equation 1)
$$\frac{x^2}{a^2} - \frac{y^2}{b^2} = c^2 \qquad \text{where} \quad a = (|GV_2| - |GV_1|)/2, \quad |GV_2| > |GV_1|,$$
$$c = |V_1V_2|/2, \quad \text{and } b^2 = c^2 - a^2.$$

If at any point the opposite-edge boundary intersects a shadow boundary, it will become defined by another hyperbola from that point on. This second hyperbola is defined by considering as foci (1) the vertex of the edge associated with the shadow boundary and which is the closer to the goal of the two vertices of that edge, and (2) the focus of the previous hyperbola which is not also a vertex of the edge associated with the shadow. The hyperbolic constant is computed as before, using the costs from the foci to the goal. The segment begins at the point where the second hyperbola intersects the first hyperbola, and continues away from the obstacle. The direction of curvature of the second hyperbola may be the same or opposite that of the first. (See Figure 17).

2. River Segments

Single isolated river segments generate four types of boundaries (see Figure 18 and Theorem V-2, Appendix A). *River-edges* are trivial boundaries (Lemma V-2.1). *Shadow* boundaries are associated with each

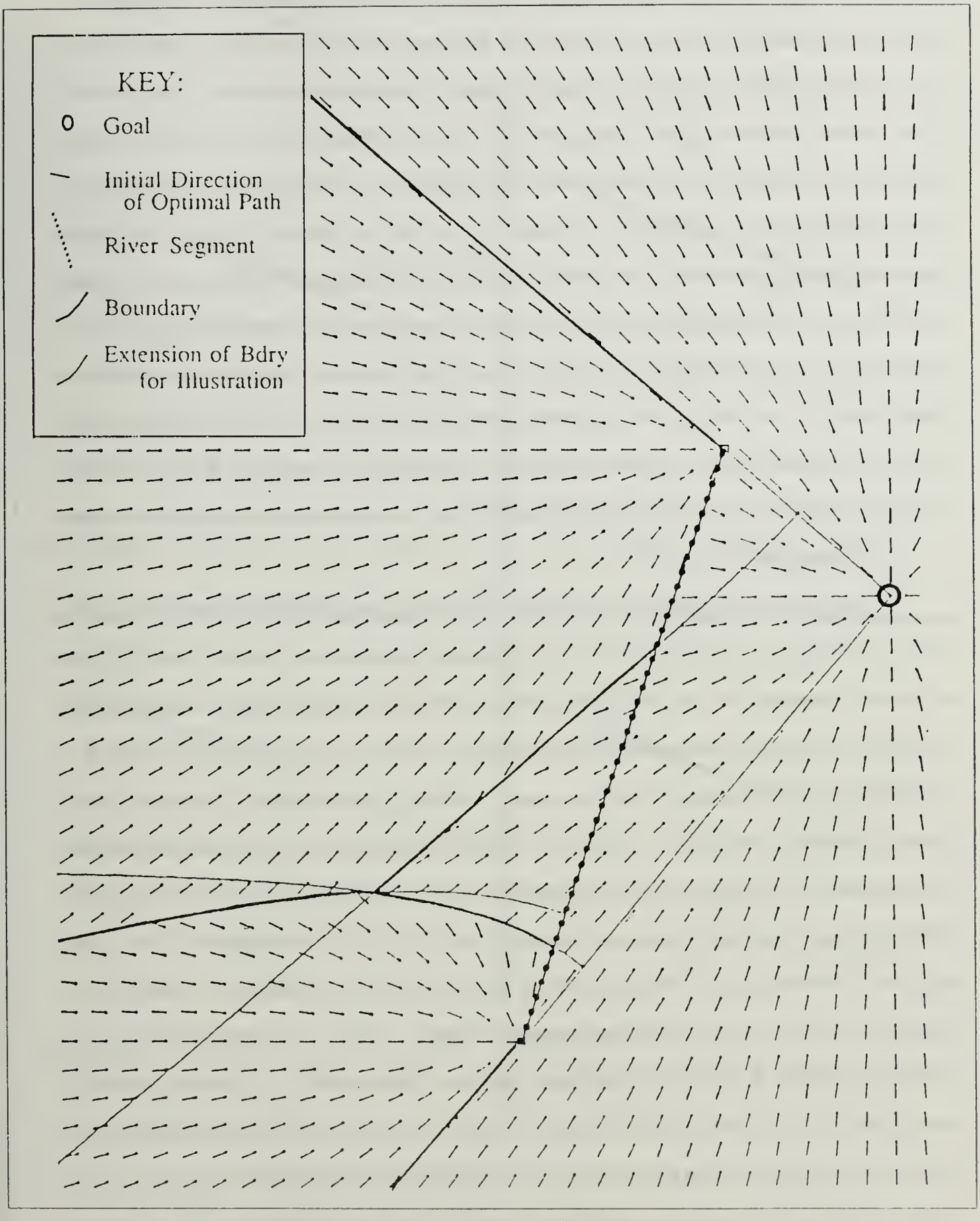


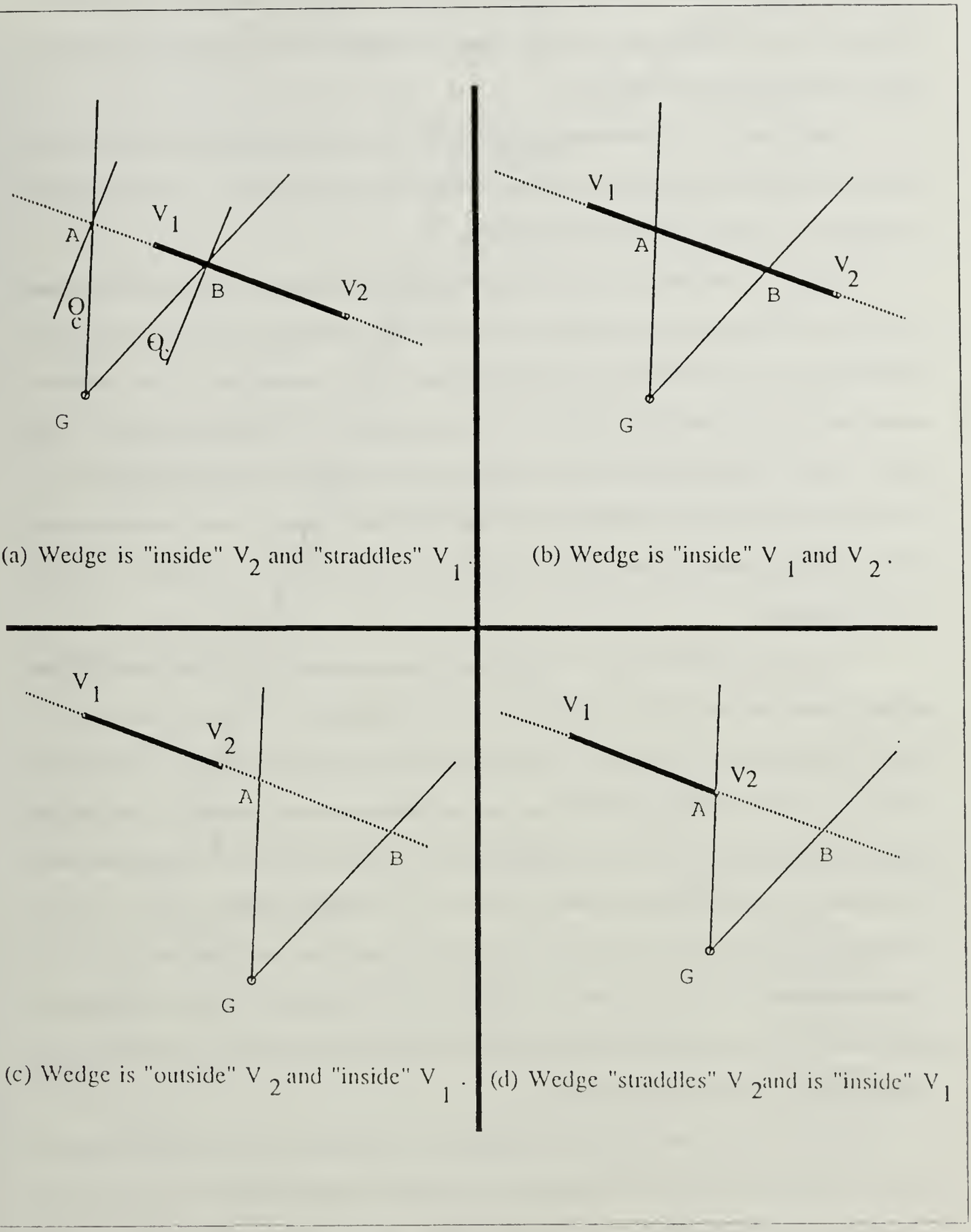
Figure 18

River Segment

river vertex, and are half-lines starting at a river vertex and lying directly away from the goal (Lemma V-2.2). *River-crossing* boundaries differentiate between paths that cross a river and ones that go around its end. A river-crossing boundary is a segment of a hyperbola defined by considering the river vertex V and the goal G as foci, with the constant in Equation 1 being a = |VG|/2. The segment begins at the point at which the hyperbola intersects the river and ends at the point at which it intersects the river-obstacle boundary (below). This type of boundary may not appear if the river-crossing cost is too high or if the angle between the river and the goal-to-river-end line approaches or exceeds 90°. (Lemma V-2.3). A river segment will act like an obstacle when the distance of the start point to the river plus the river-crossing cost is large compared with the distance from the river to the goal. If this occurs, a boundary will start at the intersection of the two river-crossing boundaries, if they exist, or if not at the river edge. This *opposite-edge boundary* will be a hyperbola defined by the two river-end vertices V_1 and V_2 as in the obstacle opposite-edge case above. (Lemma V-2.4). The river shadow boundaries will never intersect the opposite-edge boundary, so it will consist of only one hyperbola segment.

3. Road Segments

Single isolated road segments are associated with various types of boundaries, depending on their orientation with respect to the goal (Theorem V-3, Appendix A). Consider a wedge with the goal G as the vertex, formed by extending two rays from G through the line of the road intersecting the line at two points A and B, so that the interior angles GAB and GBA are the angle $\gamma_c = \pi/2 - \theta_c$, θ_c the *critical angle* such that $\theta_c = \sin^{-1}(R/S)$, for R the road cost, and S the cross-country cost, where R is greater than S. Call this the *characteristic wedge* of the road segment. (See Figure 19.) We adopt the convention for the following discussion that the wedge intersection points A and B are labelled such that their relative positions on the road line are the same as the relative positions of the two road vertices V_1 and V_2 (e.g., if V_1 is to the right of V_2 on a certain map, then A is to the right of B). When A and B and V_1 are arrayed along the road line in the order B,A,V_1, (irrespective of V_2's position), say that the characteristic wedge is *inside* V_1. When they are arrayed in the order B,V_1,A or when A and V_1 are the same point, say that the wedge *straddles* V_1. When they are arrayed in the order V_1,B,A, say that the wedge is *outside* V_1. There are seven types of boundaries induced by road segments, as listed below. When the characteristic wedge is inside V_i, types a,b,c, and d exist on the V_i end of the road segment. When the characteristic wedge straddles V_i, types a and g exist on the V_i end. When the characteristic wedge is outside V_1, types a,d, and f exist on the V_2 end, and vice versa. When the characteris-



(a) Wedge is "inside" V_2 and "straddles" V_1.

(b) Wedge is "inside" V_1 and V_2.

(c) Wedge is "outside" V_2 and "inside" V_1.

(d) Wedge "straddles" V_2 and is "inside" V_1

Figure 19

Characteristic Wedges

tic wedge is inside both V_1 and V_2, type e also exists on each end. When the characteristic wedge is inside V_1 and straddles V_2, type e exists on the V_1 end only. Figure 20 and Figure 21 show two example road segments with their associated boundaries (labeled a).

Type a: *Road-edge* boundaries separate paths which start on one side of a road from those which start on the other side. All road segments will constitute road-edge boundaries (Lemma V-3.1). For example, the road segment V_1V_2 in Figure 20 is a road-edge boundary.

Type b: *Road-end/road-travelling* boundaries separate paths which go to a road end and begin using the road from those which go to a road interior point and begin using the road. They are linear, and form a fan-shaped region at the road end. When the characteristic wedge is inside a road-end vertex V, there will be two road-end/road-travelling boundaries beginning at V and forming angles of $\pi/2-\theta_c$ and $\theta_c-\pi/2$ with the road. (Lemma V-3.2). Figure 20 shows four such boundaries (labeled b), two each emanating from road vertices V_1 and V_2, because the characteristic wedge is inside both V_1 and V_2. Figure 21 shows two road-end/road-travelling boundaries emanating from vertex V_2, because the wedge is inside V_2, but none from V_1 because the wedge is outside V_1.

Type c: *Road-end/goal* boundaries separate paths which travel directly to the goal from paths that travel to a road end and then along the road. These boundaries are segments of hyperbolas where road-end V and goal G are the foci, and the hyperbola is described by Equation 1, where V_1=G and V_2=V. The boundary begins at the point where the hyperbola intersects the road-end/road-travelling boundary. A road-end/goal boundary exists on the goal side of the road segment for vertex V_1 if and only if a pair of road-end/road-travelling boundaries exist; if the characteristic wedge is outside V_2, a road-end/goal boundary will also exist on the far side of the road segment for vertex V_1. (Lemma V-3.3). In Figure 20, two such boundaries exist (labeled c), one associated with each vertex of the road segment, and both on the goal side of the road, although the boundary on the V_2 end is not shown being off the page to the bottom. In Figure 21, two such boundaries exist associated with V_2, although both are off the page.

Type d: *Near-side road-travelling/goal* boundaries lie on the near side of the road (i.e., on the goal side) and separate paths which enter a road interior, travel along it, and then exit the road to cut over to the goal from those which go directly to the goal. These boundaries are described by segments of parabolas defined for road-end vertex V_1 as follows: the focus of the parabola is the goal, G, and the directrix is the line perpen-

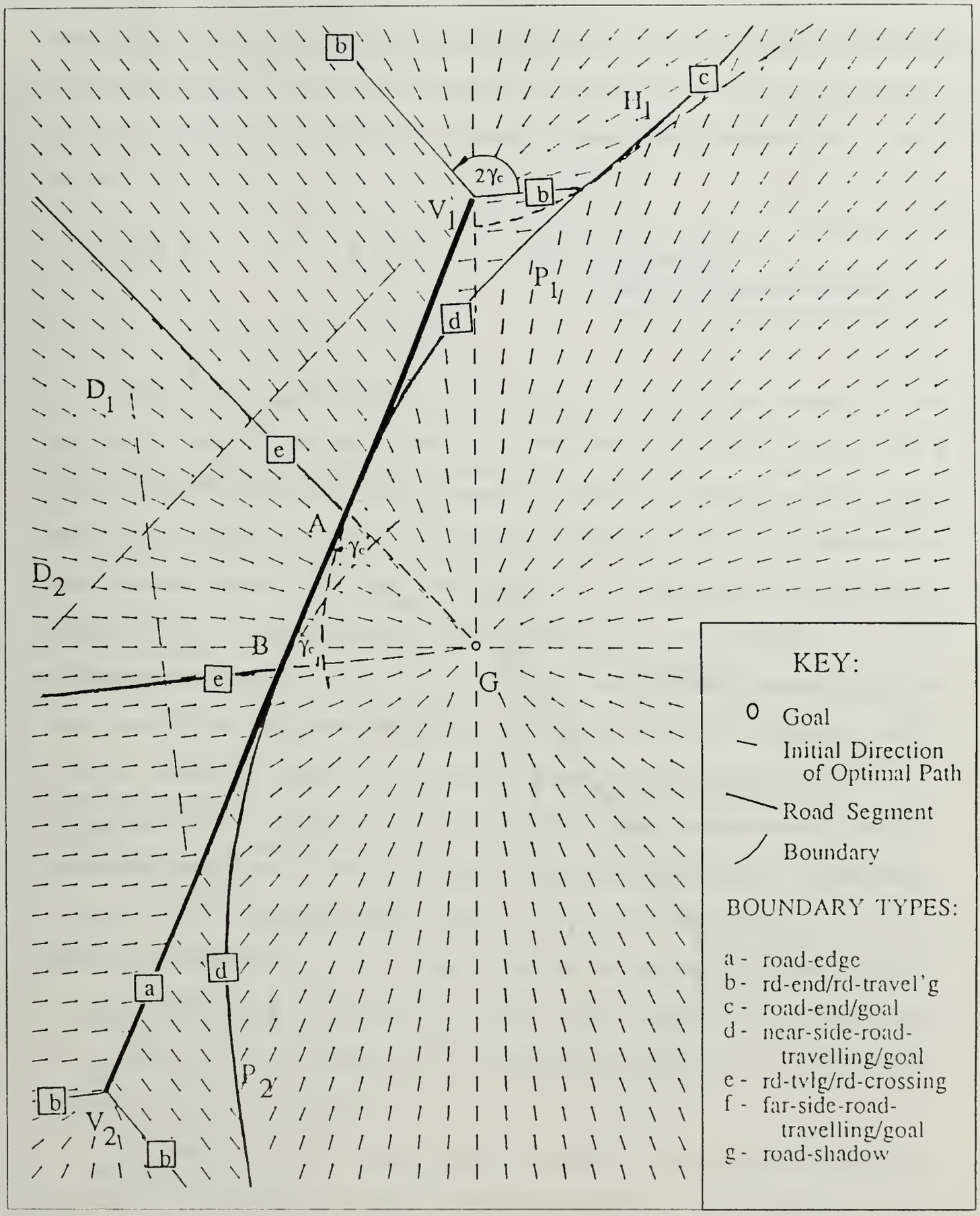


Figure 20

Road Segment Example 1

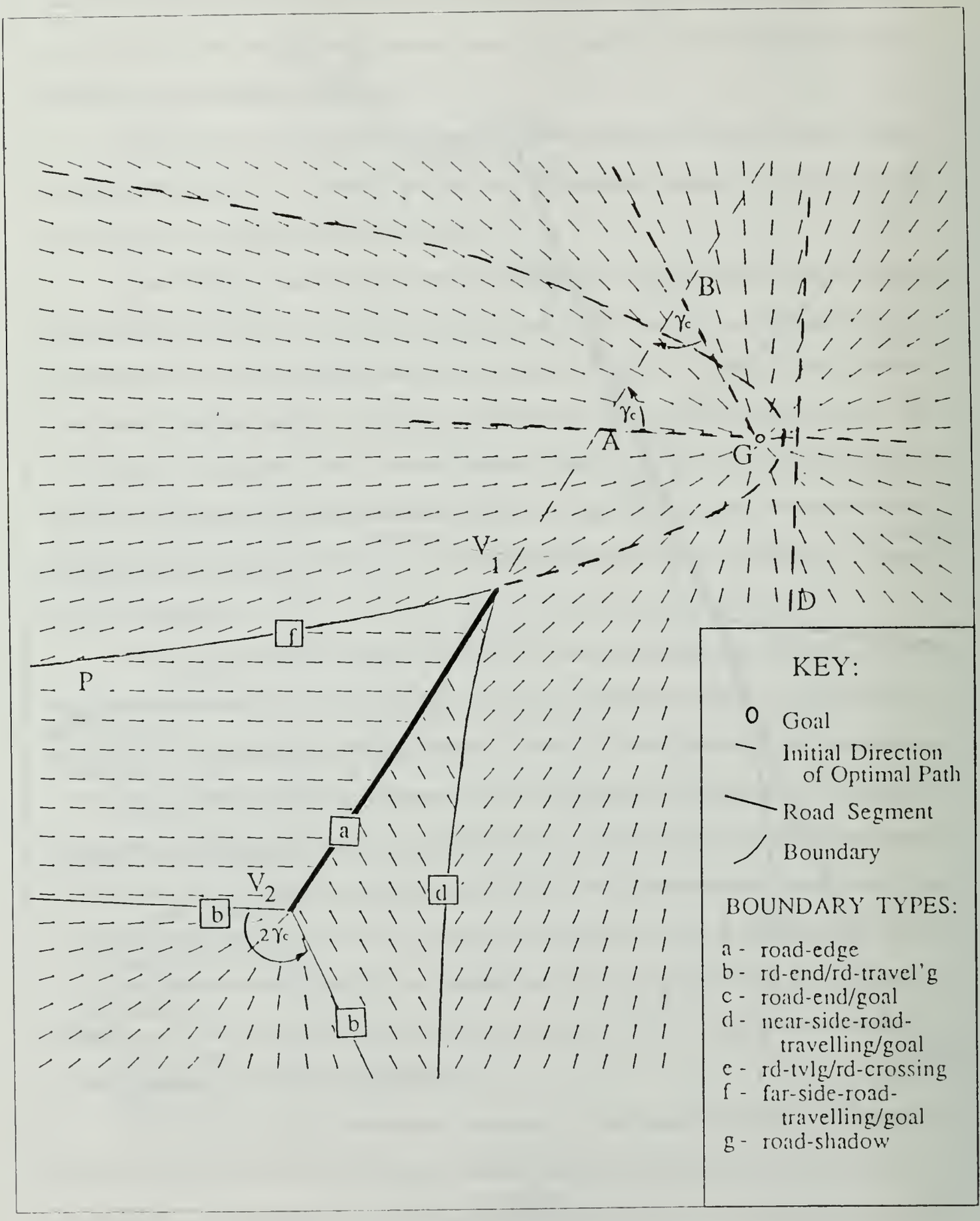


Figure 21

Road Segment Example 2

dicular to the characteristic wedge ray GB and which intersects the ray GB, and is a distance |GA| from A if the characteristic wedge is inside V_1 and not outside V_2, and a distance $|GV_2|$ from V_2 if the wedge is inside V_1 and outside V_2. This parabola is described by Equation 2, where the y-axis is the directrix and the x-axis is the axis of the parabola.

(Equation 2) $y^2 = 4 p x$ where $p = d \cos^2(\theta_c)/4$
for $d = |GA|$ if wedge not outside V_2,
and $d = |GV_1|$ if wedge is outside V_2.

The segment of the parabola which is a boundary begins at point A if the characteristic wedge is inside V_1 and not outside V_2, and begins at point V_1 if the characteristic wedge is inside V_1 and outside V_2. It ends at the point where the parabola intersects the near-side road-end/road-travelling boundary and the road-end/goal boundary if there is a road-end vertex, and continues indefinitely if there is not. It exists under the same conditions as these two exist. (Lemma V-3.4). Figure 20 shows two near-side/road-travelling boundaries (labeled d), because the wedge is inside both V_1 and V_2. The directrices D_1 and D_2 are distances |GA| from A and |GB| from B respectively, because the wedge is inside both V_1 and V_2. If the wedge had straddled either vertex, the same distances would continue to apply. Figure 21 shows one near-side/road-travelling boundary, but this one has a directrix (not shown) with a distance $|GV_1|$ from V_1 because the wedge is outside V_1.

Type e: *Road-travelling/road-crossing* boundaries separate paths which begin on the far side of the road from the goal and travel along the road from those which also begin on the far side but cross the road and go directly to the goal. This type of boundary will exist for road-end V_1 when the characteristic wedge is inside V_1 and not outside V_2. It is linear (a ray), and is the portion of the characteristic wedge ray beginning at A and lying on the far side of the road. (Lemma V-3.5). Figure 20 shows examples of two road-travelling/road-crossing boundaries which occur because the characteristic wedge is inside both vertices. Figure 21 has no such boundaries, because the wedge is outside V_2.

Type f: A *far-side road-travelling/goal* boundary occurs on the V_1 end when the characteristic wedge is outside V_1. It is a segment of a parabola with focus G and directrix D such that D is perpendicular to the ray GA, but does not intersect it (i.e., D lies on the other side of G from A), and D_2 is distance $|V_1G|$ from V_1. This parabola is defined similarly to the one in Equation 2, except that $d=|V_1G|$. One far-side road-travelling/goal

boundary occurs in Figure 21 on the V_2 end of road segment because the characteristic wedge is outside V_2. Non occurs in Figure 20, because the wedge is outside neither vertex.

Type g: A *road-shadow* boundary occurs when the characteristic edge straddles a vertex V. It separates points whose paths cross the road en route to the goal from those which go directly to the goal. The shadow boundary is a ray starting at V and lying directly away from G. (Lemma V-3.7). Note that since paths which cross roads pay no additional cost, this type of boundary occurs only by convention. We want path descriptions to reflect each terrain-feature-edge crossing, even though no change in direction or cost rate occurs for this case. This type is not illustrated in the accompanying figures.

4. Homogeneous-Cost Areas (HCA)

Homogeneous-cost areas (HCA) generate boundaries both inside and outside the HCA. The outside boundaries are similar, although not identical, to those associated with obstacles, rivers, and roads. This is not surprising, since the HCA is a generalization of each of these types of terrain. There are four cases, based on the relative costs of the HCA interior and exterior and the location of the goal inside or outside the HCA. We first consider the case where the cost of the interior of the HCA is greater than the cost of the exterior and the goal point lies outside the HCA, then the high-cost, interior-goal case, the low-cost exterior-goal case, and the low-cost interior-goal case.

a. High-Cost HCA With An Exterior Goal

When the goal is exterior to the homogeneous-cost area, and the cost of the HCA is greater than the surrounding terrain, boundaries occur according to Theorem V-4, Appendix A. Define a *visible edge* of an HCA to be an HCA edge for which no point on the edge has an optimal-path list whose first element lies on the HCA perimeter. Define a *hidden edge* as a non-visible edge, i.e., an edge for which some point on the edge has an optimal-path list whose first element lies on the HCA perimeter. Thus a hidden edge may have points whose optimal paths travel through the HCA, which would mean that their optimal paths would have as their first element the visible edge across which they pass. Define *opposite-edge sequence* as the smallest connected sequence of hidden edges for which the first and last endpoints of the edge sequence have optimal paths whose initial directions follow the HCA edges in opposite (i.e., clockwise versus counterclockwise) directions. If no such endpoint can be found at one end or the other of the sequence of hidden edges, let the endpoint at that end be the "outer" vertex of the last hidden edge, i.e., the vertex which joins the last hidden edge in the clockwise

(or counterclockwise) direction with the first visible edge in the clockwise (or counterclockwise) direction. In Figure 22, the initial direction of optimal paths for each edge endpoint is shown as a vector. HCA 1 has opposite-edge sequence ED, HCA 2 has opposite-edge sequence EDCB, HCA 3 has opposite-edge sequence FED, and HCA 4 has opposite-edge sequence JIHFE. Essentially, this definition specifies the range over which a search must be conducted for an opposite point, if one exists, and defines the HCA vertices which may generate opposite-edge boundaries (see below). Define the *opposite point* of an HCA as a point with two optimal paths lying in opposite directions (i.e., clockwise and counterclockwise) along HCA edges. If "shortcutting" occurs through the center of HCA, the opposite point might not exist, as in HCA 2 and HCA 3 of Figure 22.

Define the *critical angle* θ_c of an HCA as $\sin^{-1}(c_1/c_2)$ where the c_i are the unit costs inside and outside the HCA, and $c_1 > c_2$. An optimal path crossing an HCA edge will obey an analogue of Snell's Law in optics [Ref. 20] (see Chapter II, Section E) so that for angle of incidence θ_1 and angle of refraction θ_2, and cost rates c_1 and c_2 on either side of the edge, $c_1 \sin(\theta_1) = c_2 \sin(\theta_2)$. (See also Chapter II, Section E2b(3) and Figure II-8).

Inside a high-cost HCA with external goal, there are four types of boundaries (See Figures 23, 24, and 25). Each pair of HCA edges is potentially associated with an interior boundary. The boundary type depends on whether the edges are visible or hidden, and are on the same or opposite sides of the opposite-edge boundary. A *visible-edge* boundary distinguishes optimal paths which go through two different visible edges; the optimal paths cross their respective edges according to Snell's Law. Lemma V-4.1 (Appendix A) states the analytic form of such a boundary. Although not expressible in closed form, the boundary has much the same shape as a hyperbola segment which forms an obstacle opposite-edge boundary, i.e, it has positive but decreasing curvature from its point of incidence upon an HCA vertex inward into the HCA, and this curvature is typically small so that the curve is almost linear. An example of a visible-edge boundary is found in Figure 23, labeled a.

A *visible-hidden-edge* boundary distinguishes optimal paths going through a visible edge from those going through a hidden edge; the latter paths traverse the HCA edge at exactly the critical angle and the follow the edge. Lemma V-4.2 states the analytic form of this type of boundary, which again is similar to a hyperbola. Examples of this type of boundary occur in Figures 23, 24, and 25 and are labeled b.

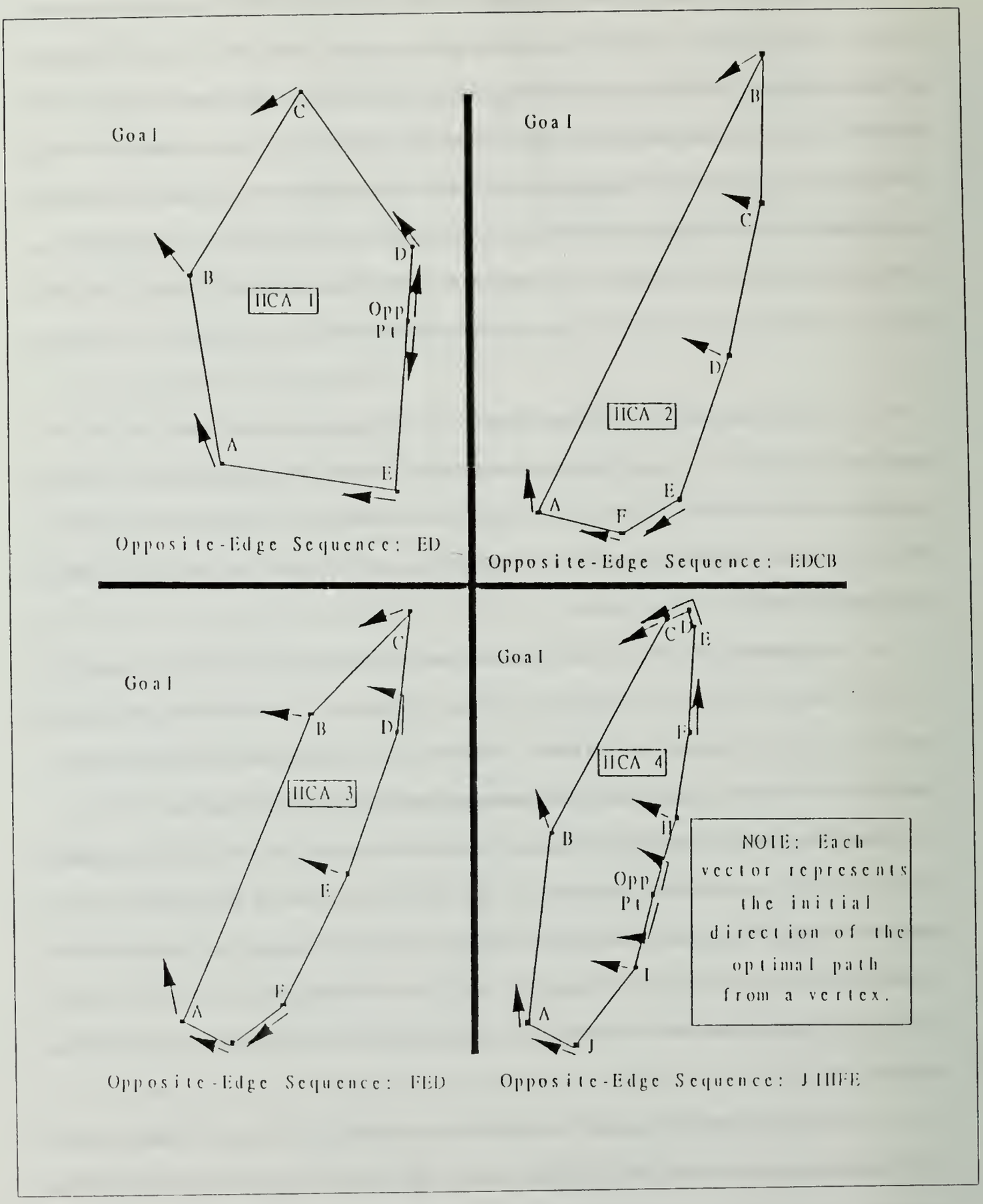


Figure 22

HCA Opposite-Edge Sequences

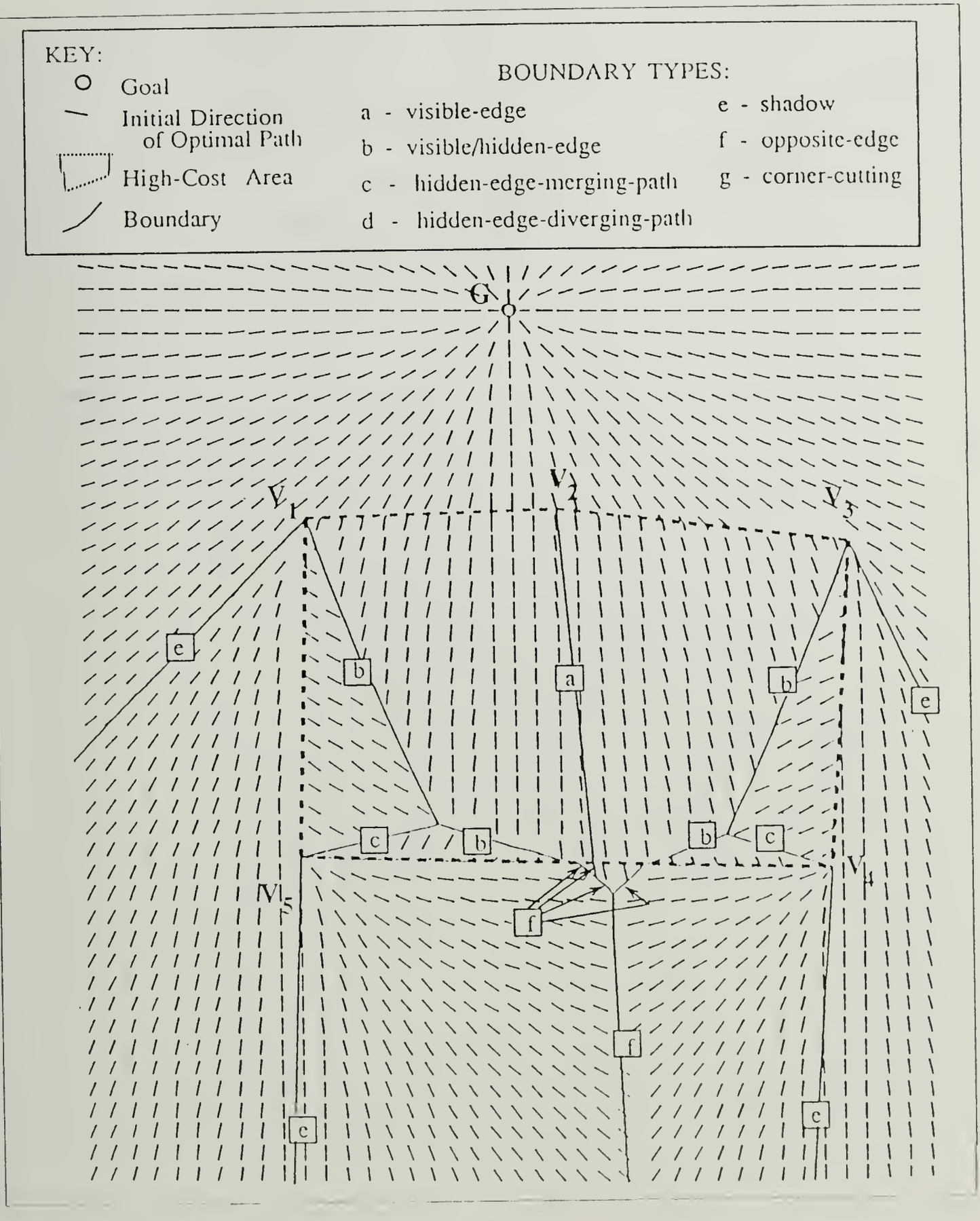


Figure 23

High-Cost, Exterior-Goal HCA Example 1

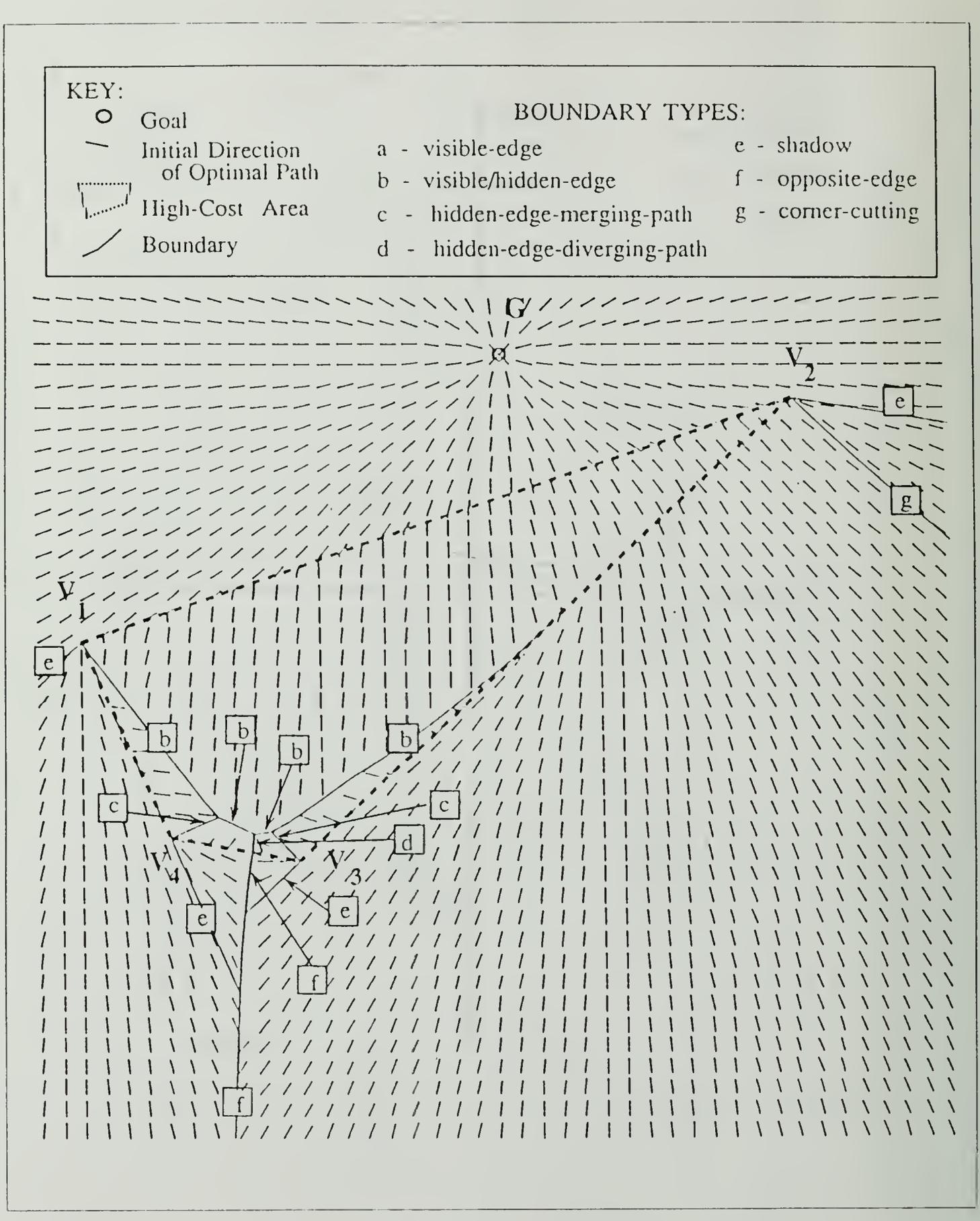


Figure 24

High-Cost, Exterior-Goal HCA Example 2

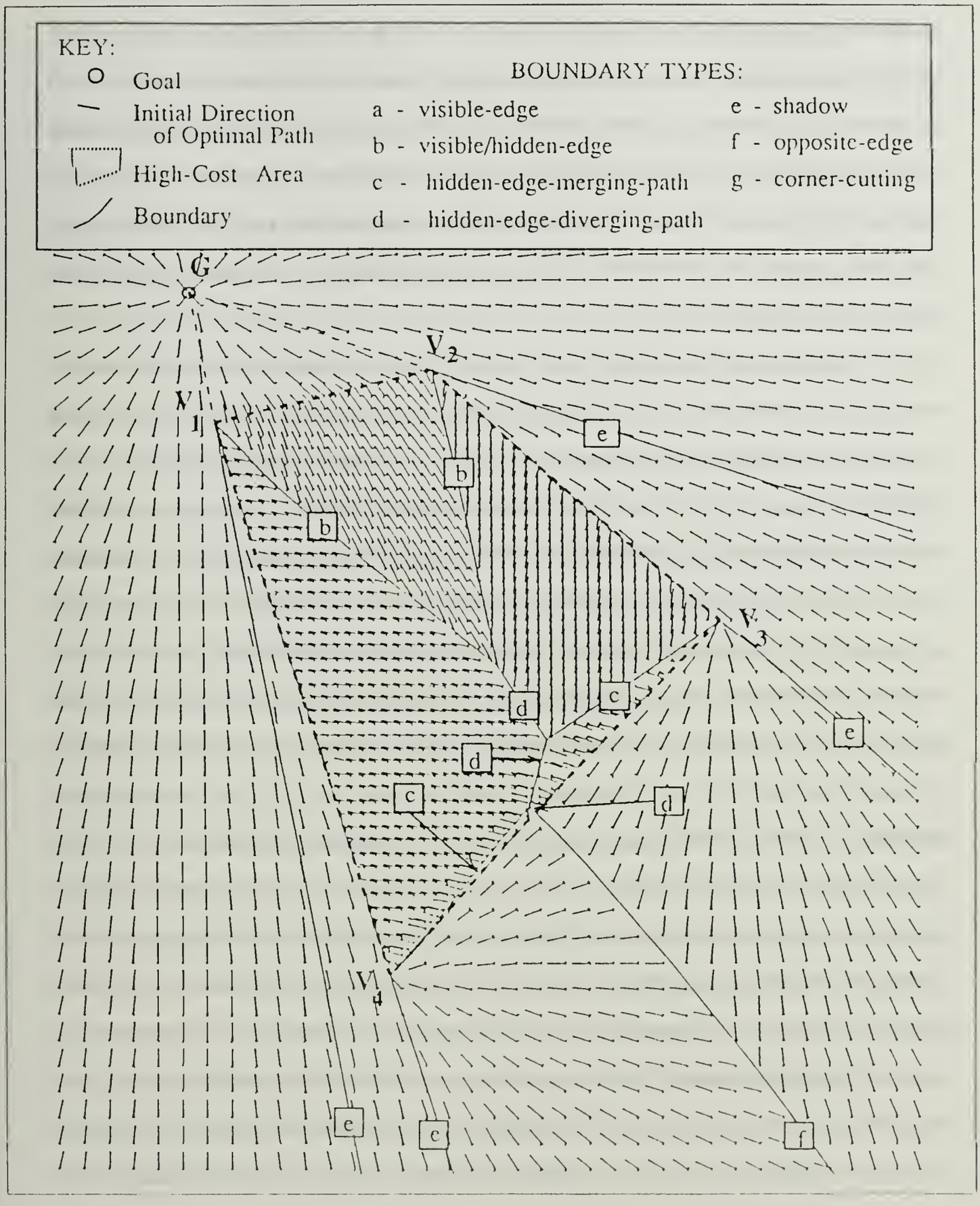


Figure 25

High-Cost, Exterior-Goal HCA Example 3

A *hidden-edge merging-path* boundary distinguishes optimal paths leaving the HCA at two different hidden edges at exactly the critical angle, and for which all paths merge before the goal. A way to check for this behavior is to see if an optimal path from a vertex of one of the edges includes a vertex of the other edge. Lemma V-4.3 states the analytic form of this type of boundary, which is a line segment. The boundaries labeled c in Figures 23, 24, and 25 are hidden-edge merging-path boundaries. A *hidden-edge diverging-path* boundary is like the preceding except the two classes of paths merge only at the goal. This type of boundary is also a line segment, as stated in Lemma V-4.4. Examples of this type of boundary occur in Figures 24 and 25 and are labeled d.

Each pair of adjacent edges is always associated with one of the above interior boundaries, while non-adjacent edges may or may not be. If shortcutting does not occur across an HCA corner, a boundary will start at the vertex at that corner. If shortcutting does occur, the boundary associated with that vertex will intersect the HCA edge at the point where shortcutting starts (see Figure 23 where two of the boundaries labeled b intersect the opposite edge, Figure 24 where one of the boundaries labeled b intersects the lower right edge of the HCA, and Lemma V-4.5). From the vertex or shortcutting point at which such a boundary begins, it will continue into the HCA interior until it intersects another boundary or HCA edge. At the point at which two such boundaries first intersect they will terminate, and a third boundary will begin which represents the division between the two regions which the first two boundaries did not have in common. For example, in Figure 23 the boundary associated with vertex V_1 distinguishes paths which cross edge V_1V_2 from those which travel along edge V_1V_5, while the boundary associated with vertex V_5 distinguishes those which travel along edge V_1V_5 from those which travel along edge V_4V_5 passing through vertex V_5. These two boundaries begin at their respective vertices and intersect in the HCA interior, and from that point a third boundary begins which distinguishes paths which cross edge V_1V_2 from those which travel along edge V_4V_5 passing through vertex V_5. These two descriptions ("crossing V_1V_2" and "travelling along V_4V_5 through V_5") represent the two regions which the initial boundaries did not have in common, so they characterize the third boundary. Boundaries will continue to intersect and new ones begin in the HCA interior until the boundary associated with each visible vertex is joined with one or more hidden vertices or HCA edges (Lemmas V-4.10 and V-4.11).

These networks of boundaries can be represented as trees, where each boundary is considered a node, and edges connect nodes whose boundaries intersect (see Lemma V-4.10). Such a tree, called an *interior-bound-*

ary tree, has interior nodes with exactly two children, while the root of such a tree can have zero, two, or four children. A tree whose root and sole node has zero children represents a boundary which goes from one edge of the HCA to another without intersecting any other boundaries, such as the boundary emanating from vertex V_2 in Figure 23. A boundary separates two regions, and any time two boundaries intersect it must be, as explained above, that they have one of the two regions in common. Beyond the point of intersection, the two regions they did not have in common must be separated by a boundary. Thus each time two boundaries intersect, a third must begin. We choose as leaf nodes those boundaries associated with HCA vertices, because one of these boundaries is guaranteed to exist for each vertex, and no other interior boundaries intersect it at the vertex or edge, so we can be sure that they will have no children. At the other end of such a boundary it either intersects an HCA edge, meaning its node is a root without children as described above, or it intersects two other boundaries, one of which will also be associated with an HCA vertex and so be another leaf node. If the latter is true, the boundary beginnning at the intersection point of the two leaf- node boundaries will serve as the parent node of the two boundaries. This merging of boundaries will continue until the parent node's boundary intersects an HCA edge, in which case the node is the tree's root, or until roots of two boundary trees are found to represent the same boundary, in which case the two trees can be merged into one. This is the case where a root will have four children, representing the two boundaries which intersect each end of the root's boundary. Several examples and illustrations of the construction of such interior-boundary trees are given in Chapter VI.

Outside the HCA, there are four types of boundaries Again, *HCA edges* are trivial boundaries (Lemma V-4.5). *HCA shadows* are defined exactly as for obstacles (Lemma V-4.6). Examples of HCA shadow boundaries are labeled e in Figures 23, 24, and 25. The other two types are *HCA opposite-edge boundaries* and *HCA corner-cutting boundaries*. HCA opposite-edge boundaries are the generalization of obstacle opposite-edge boundaries, and differentiate between paths which start outside the HCA and go through or around the HCA in different directions. There are three types of opposite-edge boundaries, depending on whether neither, one, or both optimal paths go through an HCA edge. A path which does not go through the HCA goes around it initially via one of its vertices. The case where neither path goes through the HCA is the same as the obstacle opposite-edge boundary case, and is described by connected hyperbola segments. The first and second cases have more complicated analytic forms, although the shape of the boundaries is very similar to hyper-

bolas. (Lemma V-4.7). In Figures 23, 24, and 25, opposite-edge boundaries are labeled f. In Figures 23 and 24 all three cases occur, while in Figure 25 the HCA is a *virtual obstacle*, that is, it appears to points outside it that it is an obstacle, so the only opposite-edge boundary it has is the third, or hyperbolic case.

HCA Corner-cutting boundaries occur when optimal paths cut into the HCA along an edge which is not part of the opposite-edge sequence. In fact, the analytic form of this boundary is just a variation of the second of the three types of opposite-edge boundaries discussed in the previous paragraph. Corner-cutting boundaries emanate from a vertex connecting a hidden and a visible edge when shortcutting occurs across those edges (for example, in Figure 24, labeled g). In the generalization of this case where the edges across which shortcutting occurs are separated by one or more edges, the corner-cutting boundary begins at the point at which the set of interior boundaries intersects the hidden edge (Lemma V-4.8).

The construction of interior-boundary trees is useful in finding exterior boundaries. There is exactly one opposite-edge or corner-cutting boundary associated with each interior tree of boundaries, and each visible HCA vertex is connected, either directly or via its interior boundary tree, to an opposite-edge or corner-cutting boundary. (Lemma V-4.11). When an interior boundary tree includes as a leaf node an interior hidden-edge-diverging-path boundary, the point at which the boundary intersects the HCA edge is connected with an exterior opposite-edge boundary. When an interior-boundary tree includes as a leaf node a point of intersection of an interior boundary and an HCA edge, but does not include an opposite point, for example, as happens three times along the hidden edge of the HCA in Figure 23, this point of intersection is connected with an exterior opposite-edge or corner-cutting boundary. When as happens to the rightmost vertex in Figure 24, a vertex is not connected with any interior boundary tree, corner shortcutting occurs and a corner-cutting boundary is connected with the corner vertex. Two HCA opposite-edge boundaries or corner-cutting boundaries may intersect each other or a shadow boundary, and if they do a third boundary begins at the point of intersection and lies away from the goal, as in the case of obstacle opposite-edges.

An optimal path will travel into a high-cost HCA from outside it only across an edge which forms an angle greater than $\sin^{-1}(2\theta c)$ with another connected HCA edge [Ref. 20]. If none of the hidden edges are associated with included angles of less than $2\theta_c$ with connected visible edges and the cost ratio and dimensions of the HCA allow, it acts exactly as an obstacle with respect to all start-points outside the HCA. Such an HCA is called a *virtual obstacle*. The HCA shown in Figure 25 is a virtual obstacle. If all the opposite-edge

and corner-cutting boundaries converge and become a single opposite-edge boundary away from the goal, the HCA becomes, for all points beyond the point of convergence, a virtual obstacle.

b. High-Cost HCA With An Interior Goal

An HCA containing the goal point and with higher cost than the surrounding terrain generates a set of exterior boundaries similar to the high-cost exterior-goal case, while the interior boundaries are reminiscent of road boundaries. The similarity to road boundaries arises because for start-points inside the HCA, it may be profitable to move away from the goal point initially in order to travel along an HCA edge in the exterior, lower-cost region, just as if there were a road segment along the HCA edge. Figure 26 illustrates the high-cost interior-goal case (see Theorem V-5).

We will define edges for this case with respect to each of its vertices, so that an edge may be defined differently for each of its endpoints. Define a *visible* edge with respect to one of its vertices V as an edge for which the optimal path from V cuts into the HCA interior at some point along the edge (either immediately from V or along the edge interior). Define a *hidden* edge with respect to V as an edge for which the optimal path from V starts along the other edge incident to V, or for which no optimal path from any point on the edge cuts directly into the HCA interior. Define an *opposite edge* as an edge which is a hidden edge with respect to both its vertices. There are four types of interior boundaries, which are line segments and parabola segments. Each HCA vertex can generate a set of boundaries. For each vertex V, if the optimal-path from that vertex consists only of the goal point, i.e, if the optimal path from the vertex goes directly to the goal, then there are no interior boundaries associated with that vertex.

If on the other hand the optimal path from HCA vertex V travels initially along an HCA edge, call the edge along which the path travels initially E_2, and call the other HCA edge incident upon V (along which the path does not travel) E_1. In this case there will be a boundary associated with vertex V which is a line segment. This boundary starts at V and separates paths which cut over to edge E_1 and go through V from those which cut over to edge E_2, bypassing V. This is a *hidden-edge boundary* as defined for the exterior-goal case above. (In Figure 26, boundaries labelled a are hidden-edge boundaries. Also see Lemma V-5.1 in Appendix A). In this case there will also be a parabolic boundary called a *hidden-edge/goal boundary*, which separates optimal paths which go directly to the goal from those which go initially away from the goal to edge E_1 and from there through V and on around the HCA, cutting back in to the goal at another point on the HCA

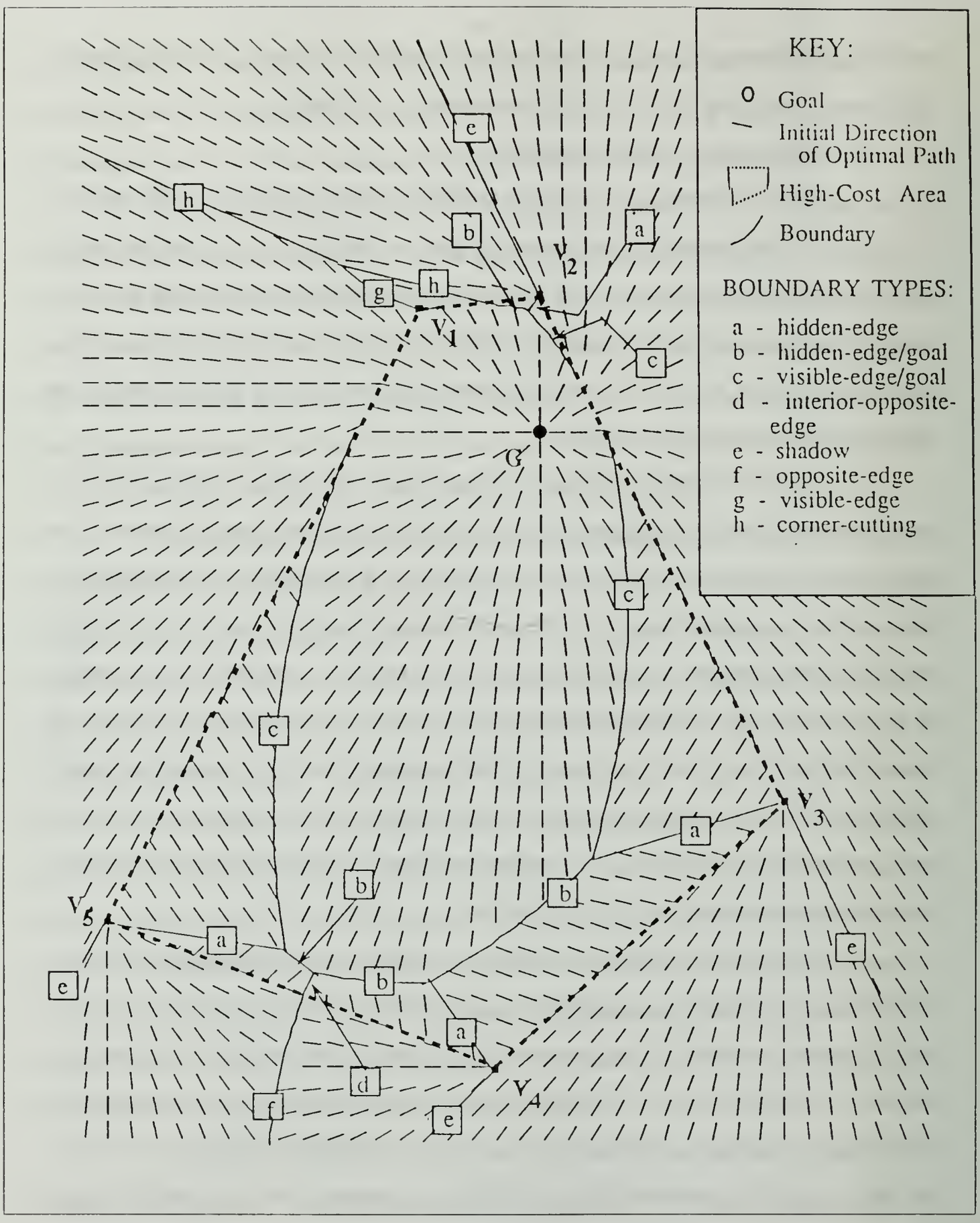


Figure 26

High-Cost, Interior-Goal HCA

perimeter. This parabola is formed by considering the goal point as the focus, and constructing the directrix such that it is perpendicular to a line from V into the HCA exterior which forms an angle of $\pi/2 + \theta_c$ with edge E_1, and such that it is a distance d from V where d = cost(optimal path from V)/c_i, where c_i = exterior cost. (See the boundaries in Figure 26 labeled b, and Lemma V-5.2, Appendix A.)

If in addition, the first turn point P on the optimal path from V is an interior point of edge E_2, i.e., if the second leg of the optimal path from V cuts into the HCA to the goal, there will be a boundary called a *visible-edge/goal* boundary associated with V and edge E_2 which separates paths that go directly to the goal from those which go initially back to E_2 then travel along E_2 to P, and then cut into the HCA at P to the goal. The visible-edge/goal boundary intersects the HCA edge at P. Again, the focus is the goal point, and in this case the directrix is perpendicular to a line from P into the HCA exterior which forms an angle with line segment PV of $\pi/2 + \theta_c$, and which is distance d from P such that d = cost(OPL(P))/c_i. (See Figure 26, the boundaries labeled c, and Lemma V-5.3.)

The other type of interior boundary occurs when two adjacent vertices on a hidden edge have optimal paths which both lie initially on an HCA edge, but which go in opposite directions around the HCA (i.e., for which neither optimal path includes the other vertex). This is the same situation that occurs in the definition of an obstacle opposite-edge, and so such an edge is called an *HCA opposite edge*. However, there may be zero, one, or more opposite edges in this case. Each HCA opposite edge V_1V_2 generates an *interior opposite-edge boundary*, which separates paths which exit the HCA and go through vertex V_1 from those which exit and go through vertex V_2. (See Figure 26, the boundary labeled c, and Lemma V-5.4.)

The exterior boundaries in this case are quite similar to the high-cost exterior-goal HCA case. There are five types of exterior boundaries. *HCA edges* are trivial boundaries (Lemma V-5.5). *Shadow boundaries* are associated with each vertex V whose optimal path OPL(V) includes as its first path-vertex a point P on the HCA perimeter. The shadow boundary is constructed by extending a ray from V along line VP away from P. (See Figure 26, boundaries labeled e, and Lemma V-5.6.)

Opposite-edge boundaries emanate from each HCA opposite edge. An opposite-edge boundary begins at an opposite point with a hyperbola segment and extends outward from the HCA, being formed exactly as in the exterior-goal case. Since there may be more than one opposite edge, there may also be more than one opposite-edge boundary. (See Figure 26, boundaries labeled f and Lemma V-5.7.) *Visible-edge* boun-

daries separate paths which cross two edges en route to the goal. This type of boundary exists whenever an optimal path from an HCA vertex goes directly to the goal. The boundary starts at the vertex and lies outward, possibly terminating when it intersects the next kind of boundary. (See boundary labeled g in Figure 26, and Lemma V-5.8.) *Corner-cutting boundaries* emanate from points at which hidden-edge/goal boundaries from the interior intersect the HCA edge. They separate points whose optimal paths cross the edge from those which go around the edge vertex. These boundaries begin at the HCA edge and are concatenated with new curve segments at each point at which the earlier curve intersects a shadow boundary, as in the corner-cutting case above. (See boundaries labeled h in Figure 26, and Lemma V-5.9.)

c. Low-Cost HCA With An Interior Goal

Analysis of an HCA with lower cost than the surrounding terrain, where the goal is in the HCA interior, shows a much simpler set of boundaries (Theorem V-6). There will never be any boundaries inside the HCA in this case, because there is no incentive for an optimal path to move away from the goal to the high-cost, external terrain, and there are no terrain-feature edges or vertices between any point in the HCA and the goal, since our HCA's are assumed convex. (See Lemma V-6.1, Appendix A.) External boundaries will occur in pairs, forming a wedge emanating from each vertex of the HCA, much as in the case of road-end/road travelling boundaries for a road segment. The external boundaries are all rays which begin at an HCA vertex and lie away from the goal, and can be constructed by tracing a path from the goal to the vertex, and then employing Snell's Law for the path with respect to each of the edges incident to the vertex to determine the orientation of the two boundaries. Call this type of boundary a *vertex/edge-crossing boundary* (see Lemma V-6.2). Figure 27 shows a low-cost HCA with interior goal, and the boundaries it induces on the plane.

d. Low-Cost HCA With An Exterior Goal

The final case, where the cost inside the HCA is lower than the surrounding terrain and the goal is outside the HCA, bears some similarities to the low-cost, interior-goal case and some to the high-cost, interior-goal case. In this case, parabolic and similar boundaries occur outside the HCA, treating HCA edges as if they were roads, and the wedges which occur in the low-cost, interior-goal case are present in this case as well. Only one type of boundary occurs in the HCA interior, and seven types occur in the HCA exterior (Theorem V-7). Figure 28 illustrates a typical low-cost, exterior-goal HCA.

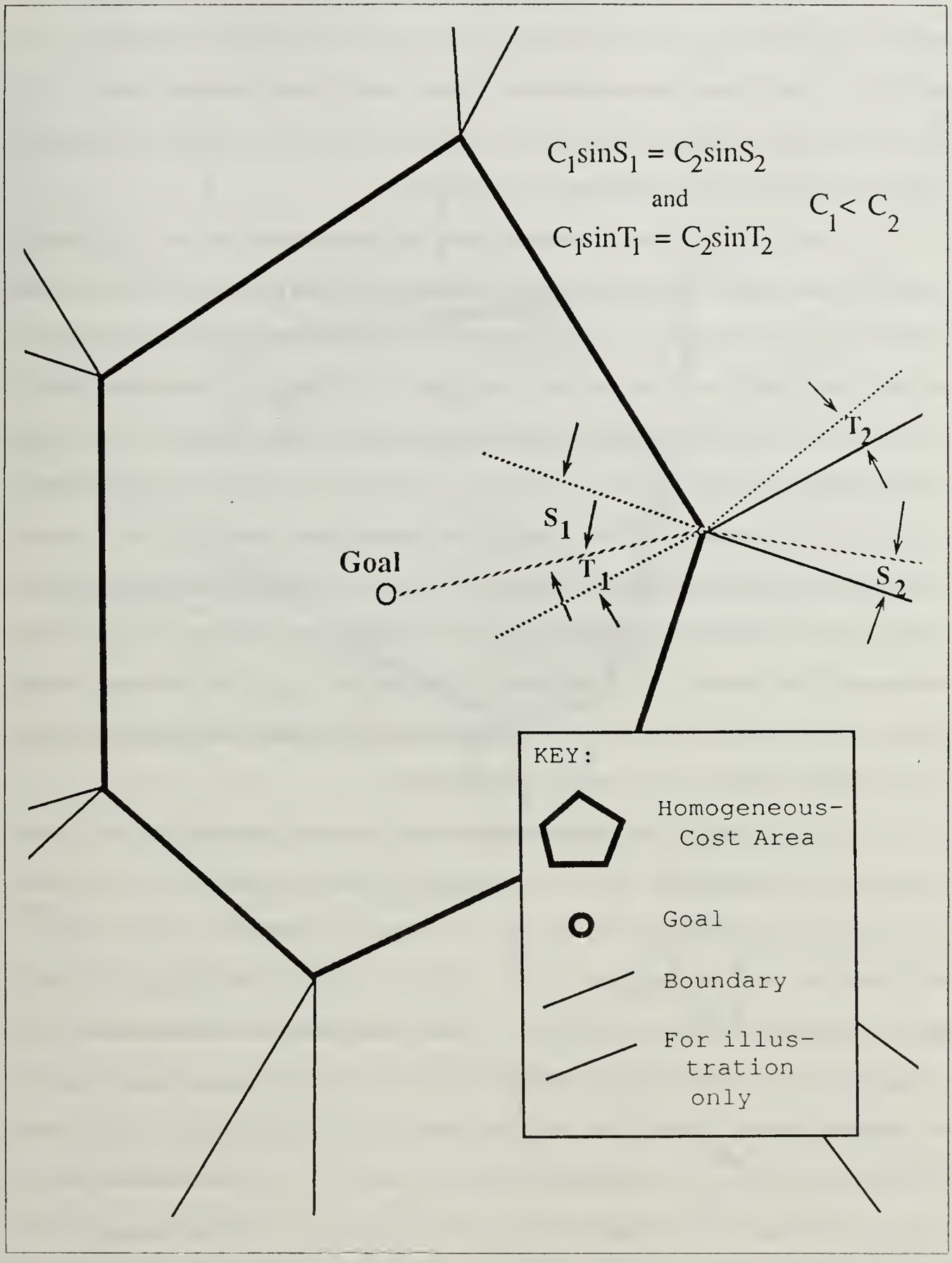


Figure 27

Low-Cost, Interior-Goal HCA

In the exterior, in addition to the trivial edge- boundaries (Lemma V-7.1), boundaries can be constructed by considering the behavior of the optimal path from each of the HCA vertices. For each vertex V of the HCA, let E_1 and E_2 be the edges incident upon V, while V_1 and V_2 are the vertices such that $VV_1 = E_1$ and $VV_2 = E_2$. Additionally, let vertex V_1 be closer to the goal than vertex V_2, i.e., the cost of the optimal path from V_1 be less than the cost of the optimal path from V_2.

If the optimal path from V goes initially along HCA edge E_1, (note that it will not go along E_2 because of the naming convention above), the paths treat the edge somewhat as if it were a road. Let P be the first point on the optimal path from V, which will be the point at which the path exits the HCA interior toward the goal. A *vertex/edge-following boundary* and a *vertex/edge-crossing boundary* are associated with from V with respect to edges E_1 and E_2 respectively. The vertex/edge-crossing boundary is a ray with vertex V lying in the HCA exterior such that the ray and the first leg of the optimal path from V form a Snell's-Law crossing of HCA edge E_2 (see Lemma V-7.2). This type of boundary separates paths which go to vertex V and then along edge E_1 from those that go directly to E_1 and follow along it. The vertex/edge-following boundary is a special case of the vertex/edge-crossing boundary where the Snell's-Law angle of the ray with edge E_1 is the critical angle θ_c (see Lemma V-7.3). These boundaries are labeled 1 in Figure 28. The vertex/edge-crossing boundary separates paths which go to a vertex V and then cut into the HCA interior from those that cross edge E_1 into the interior. In Figure 28, these boundaries are labeled 2.

Also occurring is an *edge-following/goal boundary* which is a parabola with the goal point as focus and directrix perpendicular to a line from P at an angle $\pi/2 + \theta_c$, lying a distance d away from P where d is the cost of an optimal path from P. This type of boundary separates paths which go to edge E_1 and follow the edge from those which go directly to the goal. (See Lemma V-7.4) Figure 28 has these type of boundaries labeled 4. Additionally, a *vertex/goal boundary* occurs which is similar to the road-end/goal boundary of the road segment case. This boundary begins at the point at which the edge-following/goal boundary intersects the vertex/edge-following boundary, and is a hyperbola segment with V and G being the foci, and the hyperbolic constant being the cost of the optimal path from V (see Lemma V-7.5). Figure 28 labels this type of boundary 3. This boundary may continue indefinitely, or it may intersect the vertex/edge-crossing boundary emanating from V. If these two intersect, both terminate at the point of intersection and a third boundary discussed below begins.

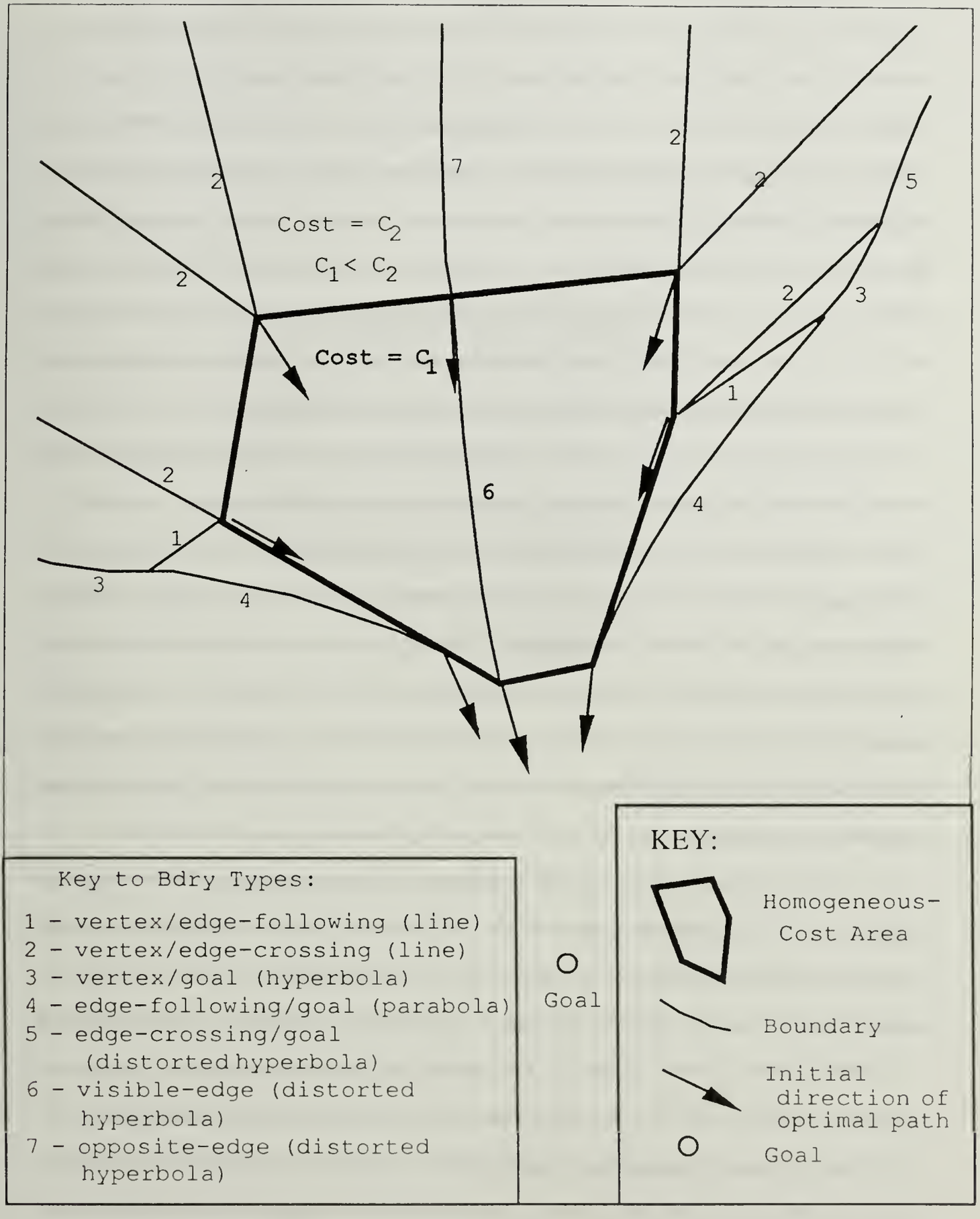


Figure 28

Low-Cost, Exterior-Goal HCA

For each HCA vertex V for which the optimal path from V goes initially into the HCA interior, a pair of linear vertex/edge-crossing boundaries will occur, just as in the interior-goal, low-cost case. These boundaries separate points whose optimal paths enter the HCA through a hidden vertex from those which enter through a hidden edge. Each boundary is constructed by extending a ray from V into the HCA exterior such that the ray and the first leg of the optimal path from V form a Snell's-Law crossing of E_1 and E_2 respectively. If in addition the vertex/goal boundary associated with vertex V_1 intersects the vertex/edge-crossing boundary emanating from V_1 associated with edge E_1, a third boundary begins. If the first point P along the optimal path from V is an HCA vertex, the boundary will be an edge-following/goal boundary, a parabola, as discussed above. If P is an interior point of an HCA edge, the boundary will a more general type of curve similar in shape to a parabola, called an *edge-crossing/goal boundary* (see Lemma V-7.6) In Figure 28, these type of boundaries are labeled 5. A vertex/goal boundary also occurs, beginning at the point at which the edge-crossing/goal boundary intersects the vertex/edge-crossing (or edge-following) boundary associated with edge E_1 . Whenever an interior boundary (see below) intersects a hidden edge of the HCA, an exterior boundary begins, called an *opposite-edge boundary* (see Lemma V-7.8 and Figure 28 boundary labeled 7). Opposite-edge boundaries separate paths which cross an edge into the HCA interior and then go across the HCA to exit across a second, visible edge, from those which cross the same first edge into the HCA but exit across a third, visible edge. Just as in the high-cost, exterior-goal case, these boundaries may intersect and new opposite-edge boundaries begin, but in this case they are of only one type and separate paths which cross one pair of edges from those which cross another pair.

There is only one type of boundary in the interior of a low-cost, exterior-goal HCA. It begins at a visible vertex which is not directly connected to any other boundaries, and separates points whose optimal paths cross one visible edge incident to the vertex from those which cross the other visible edge. Because of its similar boundary type in the high-cost exterior-goal case, it is called a *visible-edge boundary* (see Lemma V-7.7 and the boundary labeled 6 in Figure 28). Just as in that case, the interior boundaries may intersect and generate new boundaries, which are also visible-edge boundaries. Whenever a visible-edge boundary intersects a hidden edge. the visible-edge boundary terminates and an opposite-edge boundary begins in the HCA exterior. Both the visible-edge boundary and the opposite-edge boundary types are similar in shape to hyper-

bola segments, although their algebraic form is not expressible in closed form. These boundaries typically have very little curvature.

B. A UNIFYING VIEW OF REGION BOUNDARIES

The boundaries associated with each terrain feature and the homogeneous-behavior regions they separate can be viewed in a more unified manner. This view will provide the basis for a key step in the algorithm presented in Chapter VI which merges optimal-path maps for isolated terrain features into consolidated optimal-path maps.

1. Cost Functions of Regions With Respect to Region Roots

The cost of optimal paths from each start point in the plane is a function of the location of the start point. In other words, there is a cost function of X and Y which characterizes the entire map. Consider the region in the vicinity of the goal, for which the goal is the region root. Cost is proportional to distance from the goal, in the absence of intervening terrain, so iso-cost contours form circles about the goal. This cost function is an inverted cone with vertex at the goal-point, or the upper half of a cone as defined in classical geometry. In any homogeneous-behavior region with a point as its root, there will be some additional cost of the optimal path from the root to the goal. For each region whose root is a single point then, the cost function in the region will be conical with respect to a vertical axis through the point. The vertex of the cone representing the cost function will be shifted upward on the cost axis by the amount of the cost of an optimal path from the root.

Another type of region root is an edge along which paths travel en route to the goal, for example, a road segment (see Figure 29). In the discussion above regarding road segments, it was noted that the path from a point whose optimal path enters a road to travel along it does so at the critical angle $\theta_c = \sin^{-1}(C_r/C_b)$, where C_r is the cost of travelling a unit distance by road and C_b is the cost of travelling a unit distance in background terrain. Also, therefore, the cost of travelling from the point of entrance onto the road P_E to the point of exit from the road P_X is $C_r \cdot |P_E P_X| = C_b \sin(\theta_c) \cdot |P_E P_X|$.

The cost of travelling from point S to the road and along the road to the point of exit P_X is then $|SP_E|C_b + |P_E P_X|C_b \sin(\theta_c) = C_b(|SP_E| + |P_E V_X| \sin(\theta_c))$. Consider a right triangle with hypotenuse $P_E P_X$, with one leg a continuation of SP_E to the other side of the road from S to point Q. Now $|P_E Q| = |P_E P_X| \sin(\theta_b)$, so the cost of travelling from S to P_E and along the road to P_X is $C_b(|SP_E| + |P_E Q|) = C_b|SQ|$ since S, P_E, and Q are colinear.

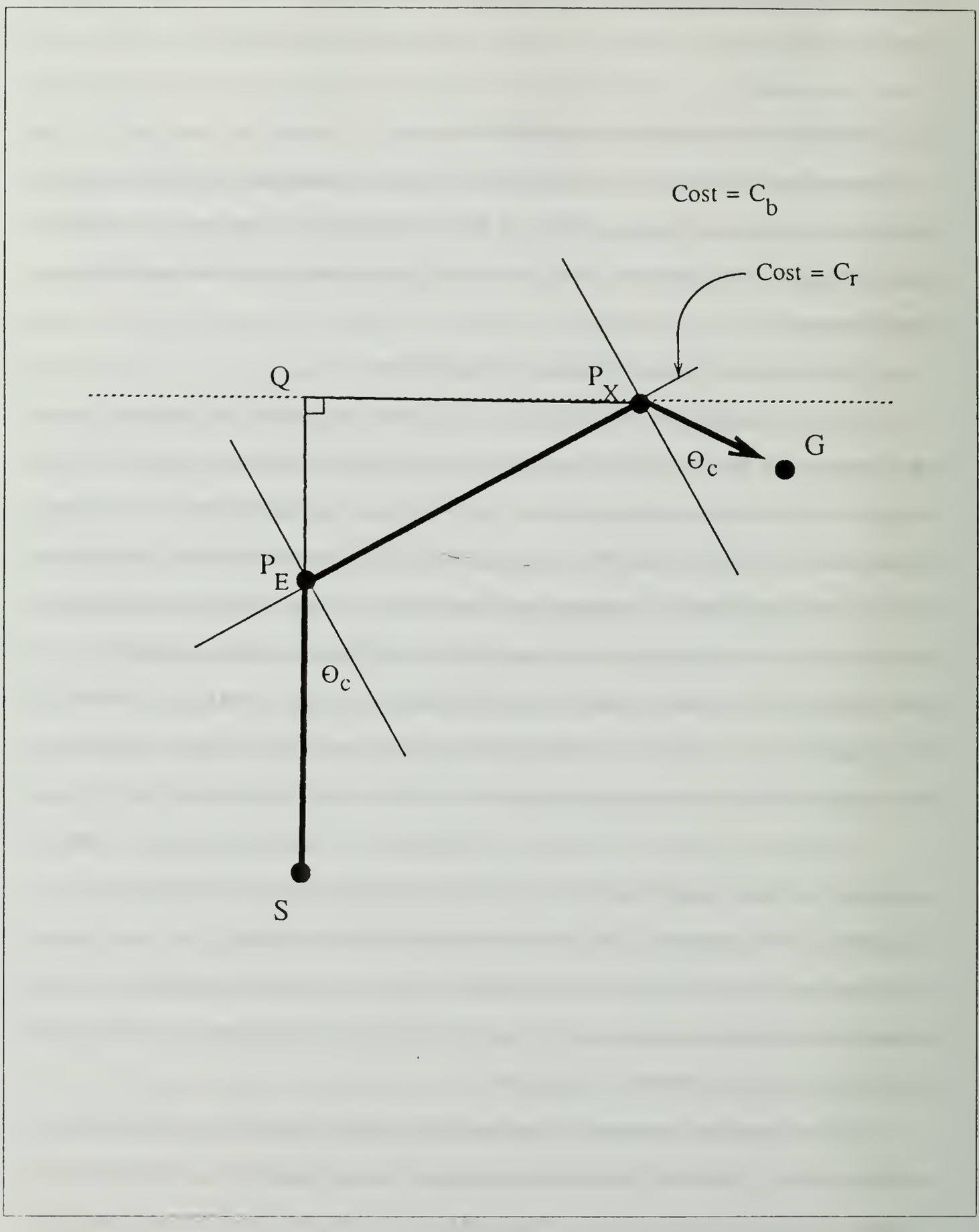


Figure 29

Cost Function for Road Segment

Thus, the cost from any point S to move to a road and travel along it to some point P_X is proportional to the distance from S to a line at angle θ_c with the road and passing through P_X. But by this description, S describes a plane which intersects line QP_X lying in the plane of the map, such that the slope of the plane in the gradient direction is $C_b|SQ|/|SQ| = C_b$. So the cost function associated with a length-wise-travelled edge is a plane.

A third type of region root is an edge which paths cross, obeying Snell's Law as they do so. As each path crosses the edge, it enters a region where the cost function becomes proportionally greater or less than before. But each edge which is crossed according to Snell's Law performs a transformation on the current cost function, or intuitively speaking, distorts the cost function. The cost function associated with a Snell's-Law edge is therefore a distortion of the cost function associated with the parent of the edge in the optimal-path tree. Thus there are two cost functions associated with Snell's-Law edges, one where the cone of a point-type root is transformed by the edge resulting in a distorted cone, and one where the plane of a road-type root is transformed by the edge, resulting in a plane. For regions with conical cost functions, paths which crossed into it from a region with a lower cost would have a cost function which was a flattened "cone". Paths crossing into it from a region with a higher cost would have a cost function which was a "cone" with greater curvature. For regions with planar cost functions, higher-cost adjacent regions would have a more sloped cost function, while lower-cost adjacent regions would have a less sloped cost function.

There are any number of "higher-order" cost functions associated with Snell's-Law edges ending in a point. For example, paths could cross three edges enroute to a point. So it does not appear to be possible to derive a finite number of analytic characterizations of cost functions for all varieties of Snell's-Law edges. Note, however, that although a cost function may be transformed by any number of Snell's-Law edges, it has its basis in either a point or a linearly-traversed edge root, so there are really only two general classifications of Snell's-Law cost functions, those for n crossings rooted in a point, and those for n crossings rooted in a linearly-traversed edge. Once a sequence of region roots leads back to a point or a traversed edge, a fixed cost is associated with the point or the goal end of the edge, which is the cost from that point to the goal, and so no other previous information about cost functions remains relevant.

A river edge can also be a region root. However, since a river edge only adds a fixed amount to the cost of paths which cross it, it serves only to shift vertically by a fixed amount whatever cost function occurs in the region on its near side, and so cannot be said to have a characteristic cost function of its own. The final

type of region root is the degenerate one, the null list, adopted by convention to represent regions which have no feasible paths, for example, obstacle interiors. Since the cost of a path in the degenerate region is infinity, the cost function will be considered undefined.

Since these are the only types of region roots which occur in the terrain defined for this research, there are only three general types of cost functions: cones, planes, and various orders of distorted cones, depending respectively on whether the region has a point as its root, a linearly-traversed-edge or one or more Snell's-Law edges ending in a linearly-traversed edge as its root, or finally a Snell's-Law edge as its root leading to one or more Snell's-Law edges and a point.

2. Boundaries Between Regions as Intersections of Cost Functions

The occurrence of many of the simpler types of boundaries can now be explained in terms of the cost functions of the region roots for regions which the boundary separates. Since at a boundary between two regions, the cost function for both regions applies, it must be that the boundary is the projection on the XY plane of the intersection of the two cost functions. The intersection of two cones with parallel axes is, according to basic analytic geometry, a hyperbola, and so it becomes clear why the boundary between two regions with points as roots is always a hyperbola.

The boundary between a region whose root is a point and a region whose root is a road-segment was determined in Section A3 above to be a parabola. Since the slope of the plane which is the cost function of the road-segment's region was shown above to be the cost rate of the background, and the slope of the cone is also the cost rate of the background, we have the condition which specifies in intersecting a plane with a cone that the intersection is a parabola.

The intersection of two planes is a line, so the boundary between regions which both have linearly-traversed edges as roots is a line segment. For example, the hidden-edge merging-path boundary of a high-cost, external-goal HCA is such a boundary, and as shown in Section A4a above is indeed a line segment.

The more complicated boundaries involving one or more Snell's-Law edges ending in a point also are consistent with this view, although the mathematics involved in computing the intersections of generalized shapes is complex. Boundaries involving Snell's-Law edges ending in a linearly-traversed edge are of the same types as those involving single linearly-traversed edges. Since there are three general types of cost functions, and each boundary can be described as the intersection of two cost functions, there are six non-redun-

dant ways that two cost functions may intersect, as in Table 5. Each entry in the last row and the last column depends on the number of edges crossed by the region root, and will be different for different numbers of edges. For some cases, a boundary listed as a parabola, hyperbola, or distorted parabola or hyperbola will degenerate to a straight line.

A view which takes into account the nature of the cost functions associated with regions which are separated by boundaries leads to a more unified approach to the derivation of the analytical forms of the boundaries. This view will become important in the process of merging several single-feature optimal-path maps discussed in Chapter VI, since there will be too many possible cases of region intersections to derive each analytical form case by case. The above six forms will provide the basis for a general solution to the problem of merging OPM's.

TABLE 5

BOUNDARY TYPES BY REGION ROOT PAIRS

	Region Root Type (cost function type)		
	point (cone)	linearly-traversed edge (plane)	S-L edge to pt (distorted cone)
point (cone)	hyperbola	parabola	distorted hyperbola
linearly-traversed edge (plane)	parabola	line	distorted parabola
S-L edge to pt (distorted cone)	distorted hyperbola	distorted parabola	distorted hyperbola

VI. ALGORITHMS FOR OPM CONSTRUCTION BASED ON SPATIAL REASONING

A. OPTIMAL-PATH TREE CONSTRUCTION

The first step in constructing an optimal-path map is to build an optimal-path tree (OPT). A straightforward way to do this is presented here, although a more efficient way would be to build the OPT during the execution of an algorithm such as recursive-wedge decomposition or the continuous-Dijkstra algorithm. A set of optimal paths from the goal point to each terrain-feature vertex is computed using any point-to-point path-planning algorithm. The turn points of these optimal paths are then sequentially inserted into the OPT by scanning each path list from the goal point to its start point as the OPT is traversed from its root (the goal) through its internal nodes, matching nodes of the tree with turn points of the path.

As the insertion algorithm traverses the OPT, a pointer identifies the current node. A pointer also identifies the current element of the path list. If the current node has a child node which matches the current element of the path list, the child node becomes the current node and the next element on the path list becomes the current one. If the current node has no child node which matches the current path-list element, a new node is created which matches the path-list element and whose parent is the current node. Then as before, the child node becomes the current node and the next path-list element becomes the current one. When the end of the path list is reached, the insertion is complete. When all the terrain-feature-vertex optimal paths have been inserted into the OPT, one final node representing the empty path list (for "start" points with no feasible paths, as for example in the middle of an obstacle) is inserted as a child of the root node and the initial OPT is complete.

B. BASIC ALGORITHMS FOR ISOLATED TERRAIN FEATURES

First, we present algorithms to construct planar partitions for four types of isolated single terrain features, given optimal-path trees. The planar partition, along with its optimal-path tree, comprises an optimal-path map. An algorithm is presented for obstacle, river segment, and road segment primitives, and for each of the four cases associated with homogeneous-cost areas (HCA).

1. An Algorithm for OPM Construction for A Single Obstacle

For a single obstacle in a homogeneous-cost background (see Figure 17), the algorithm to construct the optimal-path map with respect to a certain goal point, given the optimal-path tree, is straightforward. The OPT for a single obstacle will have three branches from the root, one of which will consist only of the empty node. Each of the other two branches will consist of one chain of nodes representing vertices on one side or the other of the obstacle. The algorithm begins by taking the obstacle edges as the starting set of homogeneous-behavior boundaries. Then it constructs all the shadow boundaries by traversing down the two branches of the optimal-path tree whose nodes represent vertices on opposite sides of the obstacle, creating a shadow boundary for each edge of the tree until it finds the leaf node of each branch. Then it constructs the opposite-edge boundary starting with the hyperbola generated by the two vertices of the opposite edge and sweeping away from the goal. Each time the current segment of the opposite-edge boundary intersects a shadow boundary, a new pair of foci is determined, and the new hyperbola segment is constructed. The algorithm is finished when the opposite-edge boundary does not intersect any more shadow boundaries.

Table 6 (on two pages) shows the algorithm for construction of a single obstacle OPM. Algorithms are presented using standard procedural conventions as in Chapter III, with natural-language explanations substituting for rigorous notation where possible without ambiguity. The input to each algorithm is a representation of the terrain feature and an optimal-path tree, representing the optimal paths from each terrain-feature vertex. The *doubly-connected-edge-list* (DCEL) data structure presented in Chapter II, Section A is used to represent the planar partition. We assume that low-level algorithms are available to manipulate the DCEL, for example, **insert-into-dcel**. Assume also that specifying an optimal-path tree node is equivalent to specifying the coordinates of the vertex represented by it, as well as the cost of the optimal path from the vertex to the goal.

The procedure **add-obstacle-opposite-edge-bdry** is called by the algorithm to construct the opposite-edge boundary as it lies outward from the obstacle. It does this by finding, if they exist, points of intersection with the shadow boundaries from the two vertices which serve as the foci of the hyperbola which is the active portion of the opposite-edge boundary, and choosing the one which occurs closest to the obstacle. Both the shadow boundary and the hyperbola are truncated at this point, and new foci and a new hyperbola are determined. This hyperbola becomes the active portion of the opposite-edge boundary, and new shadow boundaries

TABLE 6

OBSTACLE OPM ALGORITHM

```
algorithm single-obstacle-opm:                    (Algorithm VI-1)
  input: Optimal-Path Tree with root node N
    and associated obstacle edge-list O;
  output: Optimal-Path Map M  (a DCEL)
    and modified Optimal-Path Tree N;
  purpose: to construct an OPM for a single obstacle;
  {
  M := empty dcel structure;
  while (O is not empty)                          /* insert obstacle edges into DCEL. */
    {
    insert-into-dcel(M,First edge of O);
    O := O less first edge of O;
    }
  N_prev := N;                                    /* initialize N_prev to Goal. */
  j := 1;
  for each child-node of N                        /* construct shadow boundaries. */
    {
    N_curr := child-node(N);
    if N_curr has a child node                    /* ie, if node is not a leaf node. */
      {
      until  N_curr has no child nodes            /* ie, if node is a region root. */
        {                                         /* traverse to bottom of this branch. */
        N_prev := N_curr;
        Bdry := Line N_curr N_prev                /* ie, ray starting at N_curr  and */
          less ray N_curr N_prev ;                /* lying away from N_prev. */
        insert-into-dcel(M,Bdry);                 /* add shadow boundary to DCEL. */
        }
      Opposite-edge-vertex_j :=  N_curr ;         /* note: there are exactly two such vertices. */
      j := j + 1;
      }
    }
  for j := 1 to 2
    {
    Focus_j := Opposite-edge-vertex_j;
    Cost_j := cost of optimal path from
      Opposite-edge-vertex_j;
    }
   add-obst-opp-edge-bdry                         /* construct opp-edge bdry. */
    (Focus1,Cost1,Focus2,Cost2,M,N);
  }                                               /* end of single-obstacle-opm Algorithm */
```

TABLE 6 (CONTINUED)

OBSTACLE OPM ALGORITHM

```
procedure add-obstacle-opposite-edge-boundary
  input: coordinates and optimal costs from opposite-edge
    vertices, shadow bdrys represented in DCEL M, and optimal-path tree N;
  output: revised DCEL M;
  purpose: to build the opposite-edge bdry by concatenating successive hyperbola segments;
  {
  ShadBdry1 := shadow bdry from Vertex1;
  ShadBdry2 := shadow bdry from Vertex2;
  repeat until  neither shadow boundary intersects the hyperbola;
    {
    Bdry := segment of hyperbola branch such that      /*  initialize Bdry to initial leg starting at */
      Focus1 := Vertex1, Focus2 := Vertex2,            /* obstacle opposite-edge. */
      hyperbolic constant := abs(Cost1 - Cost2),
      and segment lies away from goal:
    Intersect1 := point of intersection of Bdry
      with shadow bdry from Focus1;
    Intersect2 := point of intersection of Bdry
      with shadow bdry from Focus2;
    if at least one shadow bdry intersects Bdry
      {
      j := j which minimizes length from the beginning
        of Bdry to Intersectj ;
      Bdry := portion of Bdry between its beginning
        and  Intersectj :
      insert-into-dcel(M,Bdry);                        /* add current segment of opp-e. bdry to DCEL. */
      Bdry := segment of hyperbola branch starting     /* get next segment of opposite-edge bdry. */
        at Intersectj such that Focusj := parent-node(Focusj),
        Costj = Cost of Focusj, hyperbolic constant
        := abs(Cost1 - Cost2), and segment lies away from goal;
        ShadBdryj := shadow bdry from Vertexj;         /* substitute new shadow bdry from new focus. */
      }
    else insert-into-dcel(M,Bdry);                     /* add last segment of opp-edge bdry to DCEL */
    }
  }                                                    /* end of add-obstacle-opp-edge-boundary */
```

are checked for intersections. When no intersections are found, the procedure is finished and the opposite-edge boundary is the concatenation of all the hyperbola segments.

The algorithm of Mitchell [Ref. 4] also builds an optimal-path map for an obstacle. It allows for multiple obstacles, as does Algorithm VI-1 used in conjunction with Algorithm VI-9 below, and also depends on the analytical characterization of homogeneous-behavior boundaries as line segments or sequences of connected hyperbola segments. His algorithm uses the notion of generalized visibility to build successive sub-OPMs. It merges OPMs using Voronoi-diagram construction methods from computational geometry, while algorithm VI-9 must use a more ad hoc approach, since Voronoi diagrams for the more general terrain features we consider are not available.

2. An Algorithm for OPM Construction for A Single River Segment

The algorithm to construct the planar partition for a single river segment is similarly straightforward (see Figure 18). In this case, exactly two shadow boundaries, at most two river-crossing boundaries and one opposite-edge boundary need to be constructed. If the river-crossing boundaries intersect, the opposite-edge boundary will begin at their point of intersection. Otherwise, no opposite-edge boundary will exist. A change from the obstacle algorithm is the addition to the optimal-path tree of edges which are crossed by paths, since these are homogeneous-behavior region roots. Table 7 shows the river-OPM construction algorithm.

As discussed in Chapter I, it is possible to model rivers, as well as obstacles and roads, as homogeneous-cost areas, and so Algorithms VI-4 and VI-6 could be used instead of Algorithms VI-1, VI-2, and VI-3. But these first three algorithms are simpler.

3. An Algorithm for OPM Construction for A Single Road Segment

The algorithm for a single road segment is somewhat more complicated, although still straightforward (see Table 8). As discussed in Chapter V (see Figures 19, 20, and 21), the boundaries which will exist for a road segment are determined by the positioning of the characteristic wedge with respect to the road vertices. Therefore, top-level decision logic for the algorithm is based on the position of the characteristic wedge. Procedure **construct-rd-bdry** is called to compute each specific boundary.

4. An Algorithm for A Single Convex High-Cost Exterior-Goal Homogeneous-Cost Area

The algorithm to compute the planar partition for high-cost area with an external goal is called **hca-opm-high-ext** (see Table 9). The equations for each boundary can be found in Appendix A in the Lemma cor-

TABLE 7

RIVER-SEGMENT OPM ALGORITHM

```
algorithm single-river-segment-opm                    (Algorithm VI-2)
  input: Optimal-Path Tree with root node N and associated river edge-list R with cost Cr;
  output: Optimal-Path Map M  (a DCEL) and modified Optimal-Path Tree N;
  purpose: to construct an OPM for a single isolated river segment;
  {
  M := empty dcel structure;
  for Ncurr := each river-vertex child-node of N          /* construct shadow boundaries */
      {
      Bdry := Line NcurrN less Half-line NcurrN;          /* ie, half-line starting at Ncurr, away from goal */
      insert-into-dcel(M,Bdry);                           /* add shadow boundary to DCEL. */
      }
  for j := 1 to 2
    {
    Focusj := River-Vertexj;
    Costj := cost of optimal path from River-Vertexj;
    Bdryj := segment of hyperbola branch with foci        /* river-crossing bdry for each river vertex. */
      Focusj and Goal, hyp constant = abs(Costj - Cr),
      such that branch is closer to Focusj;
    if Bdryj intersects river segment                     /* if so, find intersection point. */
      Intersectj := intersection point;
    else
      Bdryj := null list;                                 /* if not, river-crossing bdry does not exist. */
    }
  if Bdry1 is not null                                    /* (neither bdry or both bdrys will be null) */
    {
    Intersect1,2 := intersection of Bdry1 and Bdry2 ;
    Bdry1 := Bdry1 from Intersect1 to Intersect1,2 ;
    Bdry2 := Bdry2 from Intersect2 to Intersect1,2 ;
    insert-into-dcel(M,Bdry1);                            /* add river-crossing bdry 1 to DCEL. */
    insert-into-dcel(M,Bdry2);                            /* add river-crossing bdry 2 to DCEL. */
    Bdry := segment of hyperbola branch with              /* find opposite-edge bdry. */
      Focus1 and Focus2, hyperbolic constant =
      abs(Cost1 - Cost2), such that branch is closer
      to the higher-cost focus, with starting point at
      Intersect1,2, lying away from goal;
    River-edge1,2 := Line from Intersect1 to Intersect2 ;
    insert-into-opt(N,River-edge1,2);                     /* add river-crossing edge to Optimal-Path Tree. */
    }
  else
    Bdry := segment of hyperbola branch with              /* find opp-edge bdry if no river-crossing bdrys. */
      Focus1 and Focus2, hyperbolic constant =
      abs(Cost1 - Cost2), such that branch is closer
      to the higher-cost focus, with starting point at
      intersection of hyp and river, lying away from goal;
  insert-into-dcel(M,Line from Focus1 to Focus2);         /* add river edge as bdry */
  insert-into-dcel(M,Bdry);                               /* add opposite-edge bdry to DCEL */
  }                                                       /* end of single-river-opm Algorithm */
```

TABLE 8

ROAD-SEGMENT OPM ALGORITHM

```
algorithm single-road-segment-opm                    (Algorithm VI-3)
  input: Optimal-Path Tree with root node N and associated road edge-list R with cost Cr;
  output: Optimal-Path Map M  (a DCEL) and modified Optimal-Path Tree N;
  purpose: to construct an OPM for a single, isolated road segment;
  {
  M := empty dcel structure;
  θc := sin-1(Cr/Cbackground);                       /* road critical angle */
  Wedge-Ray1 := ray from G intersecting road V1V2
    at Pt A such that ∠GAV2 =π/2 − θc;
  Wedge-Ray2 := ray from G intersecting road V1V2
    at Pt B such that ∠GBV1 = π/2 − θc;              /* A is oriented to B as V1 is to V2 (see Chap V)*/
  if pts A, B, and V1 are ordered "BAV1"             /* wedge is "inside" V1 so generate boundary */
    {
    construct-rd-bdry(road-end/travelling,V1);       /* types b,c and d on the V1 end. */
    construct-rd-bdry(road-end/goal,V1);
    construct-rd-bdry(near-side-road-travelling/goal,V1);
    if pts A,B, & V2 are not ordered "V2AB"          /* if in addition wedge is not "outside" V2, */
      construct-rd-bdry(road-travelling/crossing,V1); /* generate type e bdry on V1 end. */
    }
  else if they are ordered "BV1A"                    /* wedge "straddles" V1 so generate boundary */
    construct-rd-bdry(road-shadow,V1);               /* type g on the V1 end */
  else if they are ordered "V1BA"                    /* wedge is "outside" V1 so generate boundary */
    {
    construct-rd-bdry(near-side-rd-travell'g/goal,V2); /* types d on the V2 end and f on the V1 end */
    construct-rd-bdry(far-side-road-travelling/goal,V2);
    }
  if pts A, B, and V2 are ordered "ABV2"             /* wedge is "inside" V2 so generate boundary */
    {
    construct-rd-bdry(road-end/travelling,V2);       /* types b,c and d on the V2 end. */
    construct-rd-bdry(road-end/goal,V2);
    construct-rd-bdry(near-side-road-travelling/goal,V2);
    if pts A,B, & V1 are not ordered "V1BA"          /* if in addition wedge is not "outside" V1, */
      construct-rd-bdry(road-travelling/crossing,V2); /* generate type e bdry on V2 end. */
    }
  else if they are ordered "AV2B"                    /* wedge "straddles" V2 so generate boundary */
    construct-rd-bdry(road-shadow,V2);               /* type g on the V2 end. */
  else if they are ordered "V2AB"                    /* wedge is "outside" V2 so generate boundary */
    {
    construct-rd-bdry(near-side-rd-travell'g/goal,V2); /* types d on the V1 end and f on the V2 end. */
    construct-rd-bdry(far-side-road-travelling/goal,V1);
    }
  }                                                  /* end of single-road-opm algorithm */
```

TABLE 8 (CONTINUED)

ROAD-SEGMENT OPM ALGORITHM

```
procedure construct-rd-bdry
; input: type of bdry T, vertex V of road, DCEL M, Optimal-Path-Tree N, and Wedge-Ray1 and Wedge-Ray2
  output: revised DCEL M and revised OPT N;
  purpose: construct each road-generated boundary of type T;
  {j := 3-i;                                          /* if i=1, j=2 and if i=2, j=1, i.e., j is other end */
  if T = road-end/travelling                          /* Type "b" boundary */
    {Bdry1 := Ray with vertex Vi, lying on line ViX, such that ∠VjViX = π/2+θc;
    Bdry2 := Ray with vertex Vi, lying on line ViY, such that ∠VjViY = 3π/2−θc;
    insert-into-dcel(M,Bdry1);                        /* add road-end/travelling boundary to DCEL. */
    insert-into-dcel(M,Bdry2);                        /* add road-end/travelling boundary to DCEL. */
    }
  else if T = road-end/goal                           /* Type "c" boundary */
    {Bdry := the branch closer to Vi of a hyperbola with foci Vi and Goal, and hyp. constant = cost from
      Vi to Goal via road, starting at point of intersection between hyperbola and type b bdry from Vi;
    insert-into-dcel(M,Bdry);                         /* add road-end/goal boundary to DCEL. */
    }
  else if T = near-side-road-travelling/goal          /* Type "d" boundary */
    {if (wedge is not outside Vj)                     /* wedge is inside Vi  & not outside Vj */
      Bdry := segment of parab. s.t. focus = Goal, and directrix D⊥Wedge-Rayj with D being |GP| from P
        (P=A if i=1, else P=B), starts at P, lies away from Goal, ends at inters. with type b bdry from Vi;
    else                                              /* wedge is inside Vi  & outside Vj */
      Bdry := segment of parabola with focus = Goal and directrix = line L, L⊥Wedge-Rayj such that L is
        |GVj| from Vj, starting at P := Vj, lying away from Goal, ending at inters. with type b bdry from Vi;
    insert-into-dcel(M,Bdry);                         /* add near-side-road-trvlg/goal bdry to DCEL. */
    insert-into-opt (N,PVi,Near-side);                /* add travelled road segment to OPT. */
    insert-into-dcel(M,PVi);                          /* add travelled segment as edge bdry to DCEL */
    }
  else if T = road-travelling/crossing                /* Type "e" boundary */
    {Bdry := ray starting at P and lying along Wedge-Rayi,
      (where P=A if i=1 and P=B if i=2), lying away from Goal;
    insert-into-dcel(M,Bdry);                         /* add road-travelling/crossing bdry to DCEL.*/
    insert-into-opt (N,edge PVi,Far-side);            /* add road segment which is travelled to OPT. */
    insert-into-dcel(M,PVi);                          /* add travelled segment as edge bdry to DCEL */
    }
  else if T = far-side-road-travelling/goal           /* Type "f" boundary */
    {insert-into-opt (N,ViVj,Near-side);
    insert-into-opt (N,ViVjFar-side);
    insert-into-dcel (M,line ViVj);                   /* road-edge boundary added to DCEL */
    for k := 1 to 2
      {Bdryk := segment of parabola with focus = Goal, and directrix = line L, L⊥Wedge-
        Rayk such that L is |GVi| from Vi, starting at Vi and lying away from Goal;
      insert-into-dcel(M,Bdryk);                      /* add far-side-road-trvlg/goal bdrys to DCEL. */
      }
    }
  else if T = road-shadow                             /* Type "g" boundary */
    {Bdry := ray from Vi along line ViG, lying away from Goal;
    insert-into-dcel(M,Bdry);                         /* add road-shadow bdry to DCEL. */
    }
  }                                                   /* end of construct-rd-bdry */
```

TABLE 9

HIGH-COST EXTERIOR-GOAL HCA OPM CONSTRUCTION ALGORITHM

```
algorithm hca-opm-high-ext                          (algorithm VI-4)
  input: Optimal-Path Tree N, HCA A with n vertices;
  output: DCEL M, and modified OPT N;
  purpose: to construct the OPM for a high-cost, exterior-goal HCA.
  {
  for i := 1 to n                                   /* add interior bdry for each vertex. */
    {
    if edge i is visible
      if edge i+1 is visible
        B := value returned by construct-high-ext-hca-bdry("visible-edge",i);
      else
        B := value returned by construct-high-ext-hca-bdry("visible-hidden",i);
    else if edge i is hidden
      if edge i+1 is visible
        B := value returned by construct-high-ext-hca-bdry("visible-hidden",i+1);
      else
        if edges are on different sides of opposite edge
          B := value returned by construct-high-ext-hca-bdry("hidden/diverging",i);
        else
          B := value returned by construct-high-ext-hca-bdry("hidden/merging",i);
    add B to BdrySet;
    }
  BdrySet := value returned by pair-and-merge-bdrys  /* join interior bdrys together. */
    (BdrySet,"high-ext-hca-interior");
  form BdryTrees from bdrys in BdrySet;
  for each BdryTree
    {
    find point X at which BdryTree                  /* there will be exactly one X per tree */
      intersects an opposite edge;
    B := value returned by construct-high-ext-hca-bdry("opposite-edge",X);
    add B to OEBdrySet;
    }
  for i := 1 to n
    if Vi connects a visible and a hidden edge
      {
      if Vi is not connected to any interior BdryTree
        add construct-high-ext-hca-bdry("corner-cutting",X) to BdrySet;
      B := value returned by construct-high-ext-hca-bdry("shadow",i);
      j := other vertex of Ei
      while Ej is not an opposite edge              /* work around the HCA creating shadow */
        {                                           /* bdrys until the opposite edge is found. */
        B := value returned by construct-high-ext-hca-bdry("shadow",j);
        j := other vertex of Ei
        }
      }
  B := value returned by pair-and-merge-bdrys       /* join opposite-edge bdrys together. */
    (OEBdrySet,"high-ext-hca-exterior");
  add B to BdrySet;
  for all B ∈ BdrySet
    insert-into-dcel(M,B);
  }                                                 /* end of hca-opm-high-ext */
```

TABLE 9 (CONTINUED)

HIGH-COST EXTERIOR-GOAL HCA OPM CONSTRUCTION ALGORITHM

```
procedure pair-and-merge-bdrys                    /* join connecting bdrys together. */
   input: BdrySet, and type of region;
   output: revised BdrySet;
   purpose: to take an initial set of boundaries, pair the ones which first intersect each other, and
      propagate a new bdry from each intersected pair, continuing until all appropriate bdrys are joined.
   {
   while BdrySet is changing
      {
      PairedBdrySet := BdrySet;
      while PairedBdrySet is changing
         {
         for all B_i,j ∈ PairedBdrySet where B_i,j is unmarked
            {
            discard B_i,j from PairedBdrySet;
            add B_i,j from BdrySet to PairedBdrySet;
            intersect B_i,j  with B_h,i and truncate both;
            add B_h,i^trunc and B_i,j^trunc to PairedBdrySet;
            intersect B_i,j  with B_j,k andtruncate both;
            add B_i,j^trunc and B_j,k^trunc := to PairedBdrySet;
            }
         for all B_i,j ∈ PairedBdrySet
            discard all but the shortest B_i,j from PairedBdrySet;
         unmark all bdrys in PairedBdrySet;
         for all B_i,j and B_j,k ∈ PairedBdrySet such that B_i,j adjoins B_j,k
            mark B_i,j and B_j,k;
         }
      for all B_i,jand B_j,k ∈ PairedBdrySet such that B_i,j adjoins B_j,k
         add B_i,k to PairedBdrySet;
      BdrySet := PairedBdrySet;
      }
   }                                              /* end of pair-and-merge-bdrys. */
```

TABLE 9 (CONTINUED)

HIGH-COST EXTERIOR-GOAL HCA OPM CONSTRUCTION ALGORITHM

```
procedure construct-high-ext-hca-bdry          /* provides methods to construct each type of */
   input: type of bdry T; index of vertex i;   /* bdry of high-cost, exterior-goal HCA. */
   output: Bdry, the resulting boundary;
   purpose: to construct a boundary generated by vertex i of type T;
   {
   if T = "visible-edge"
      Bdry := curve as specified in Lemma V-4.1;
   if T = "visible-hidden"
      Bdry := curve as specified in Lemma V-4.2;
   if T = "merging"
      Bdry := curve as specified in Lemma V-4.3;
   if T = "diverging"
      Bdry := curve as specified in Lemma V-4.4;
   if T = "hca-edge"
      Bdry := curve as specified in Lemma V-4.7;
   if T = "shadow"
      Bdry := curve as specified in Lemma V-4.8;
   if T = "opposite-edge"
      Bdry := curve as specified in Lemma V-4.9;
   if T = "corner-cutting"
      Bdry := curve as specified in Lemma V-4.10;
   }                                           /* end of construct-high-ext-hca-bdry */
```

responding to the boundary type (see also Figures 23, 24, and 25). Each vertex of such a HCA is associated with an internal boundary, whose character depends on whether the edges incident to the vertex are visible or hidden (and for vertices on two hidden edges, on whether the vertices nearest the goal for each edge have optimal paths which go in the same, or different directions around the HCA, called merging or diverging paths respectively). These boundaries are computed first, and then procedure **pair-and-merge-bdrys** constructs a network (or networks) of interior boundaries which is connected to the initially-computed boundaries. This procedure pairs boundaries which intersect, and then plots a new boundary which has an endpoint at the point of intersection of the paired boundaries. It continues pairing boundaries and plotting new ones until all the boundaries are joined together on both ends or intersect an edge of the HCA. Note that deciding which adjacent boundaries should be paired together is not simple, and it may take several iterations for the procedure to settle on a correct configuration.

The interior boundaries are then joined into trees, and since each interior boundary tree intersects an opposite edge exactly once, this can serve to begin generation of the external opposite-edge boundaries. In contrast to obstacles, there can be several HCA opposite edges and opposite-edge boundaries. Corner-cutting boundaries are indicated when an interior boundary associated with a vertex actually begins, not at the vertex, but somewhere along the boundary. The algorithm next checks for this situation, which can only happen with respect to a vertex joining a hidden and a visible edge. This type of vertex is also a good place to begin generating shadow boundaries. Finally, **procedure pair-and-merge-bdrys** is again used, this time with the exterior shadow and opposite-edge boundaries.

Figures 30, 31, and 32 illustrate the state of procedure **pair-and-merge-bdrys** at various intermediate stages in its execution for the example HCAs of Figures 23, 24, and 25 respectively. Edges of the HCAs are numbered, and boundaries are labeled "i,j", where i and j represent the edges crossed by paths on either side of the boundary. Boundaries which are paired with another boundary at each stage are noted by an asterisk. Boundaries which are stored in the data structure PairedBdrySet are noted in the figures as dark lines. Figure 30a, 31a, and 32a show the interior boundaries associated with each terrain-feature vertex at the beginning of the algorithm (beginning at the vertex or associated short-cutting point, extending indefinitely into the interior and then beyond. The current set of interior-boundary trees is also shown, with each node labeled by the boundary it represents.

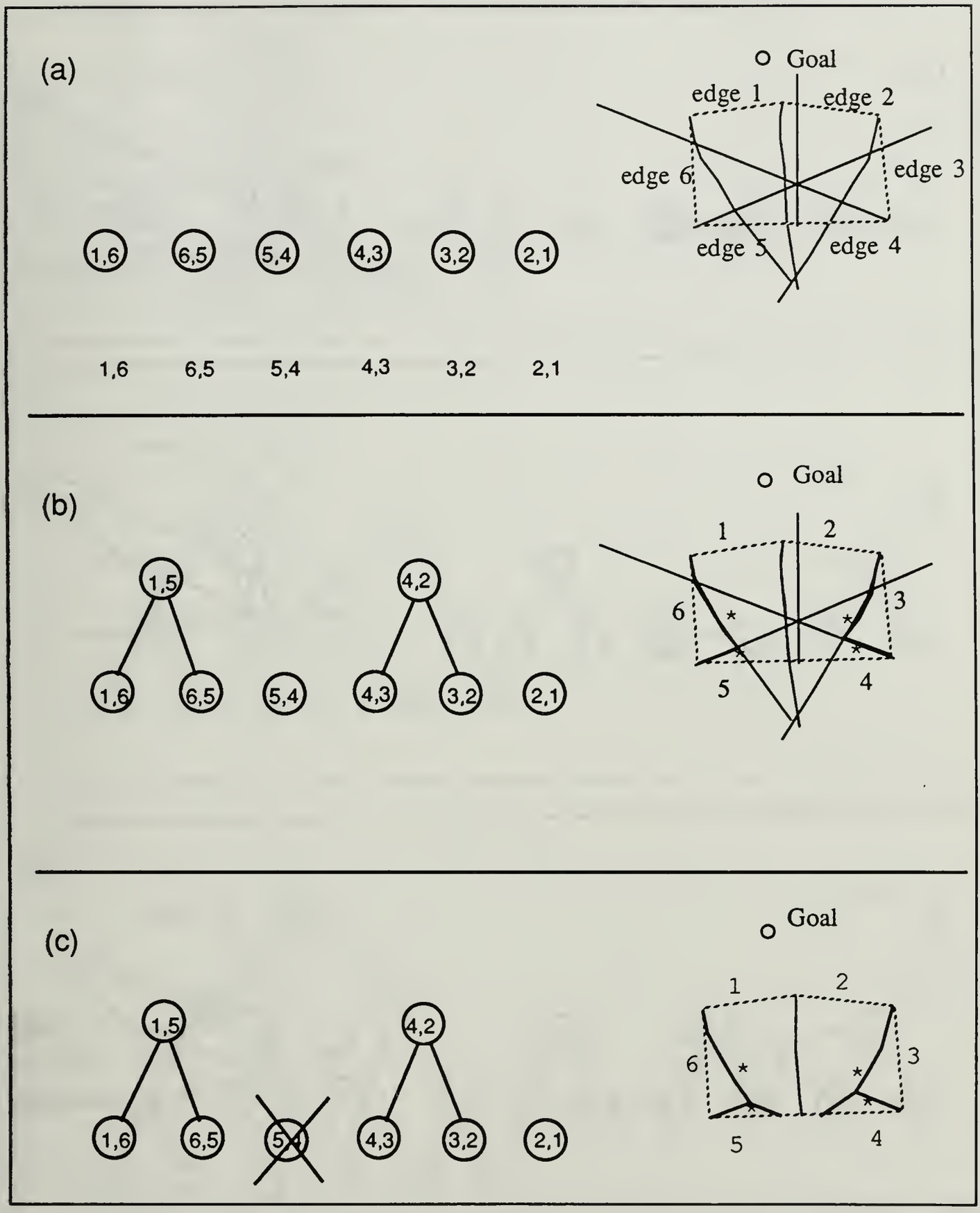


Figure 30

Construction of HCA Interior-Boundary Tree Example 1

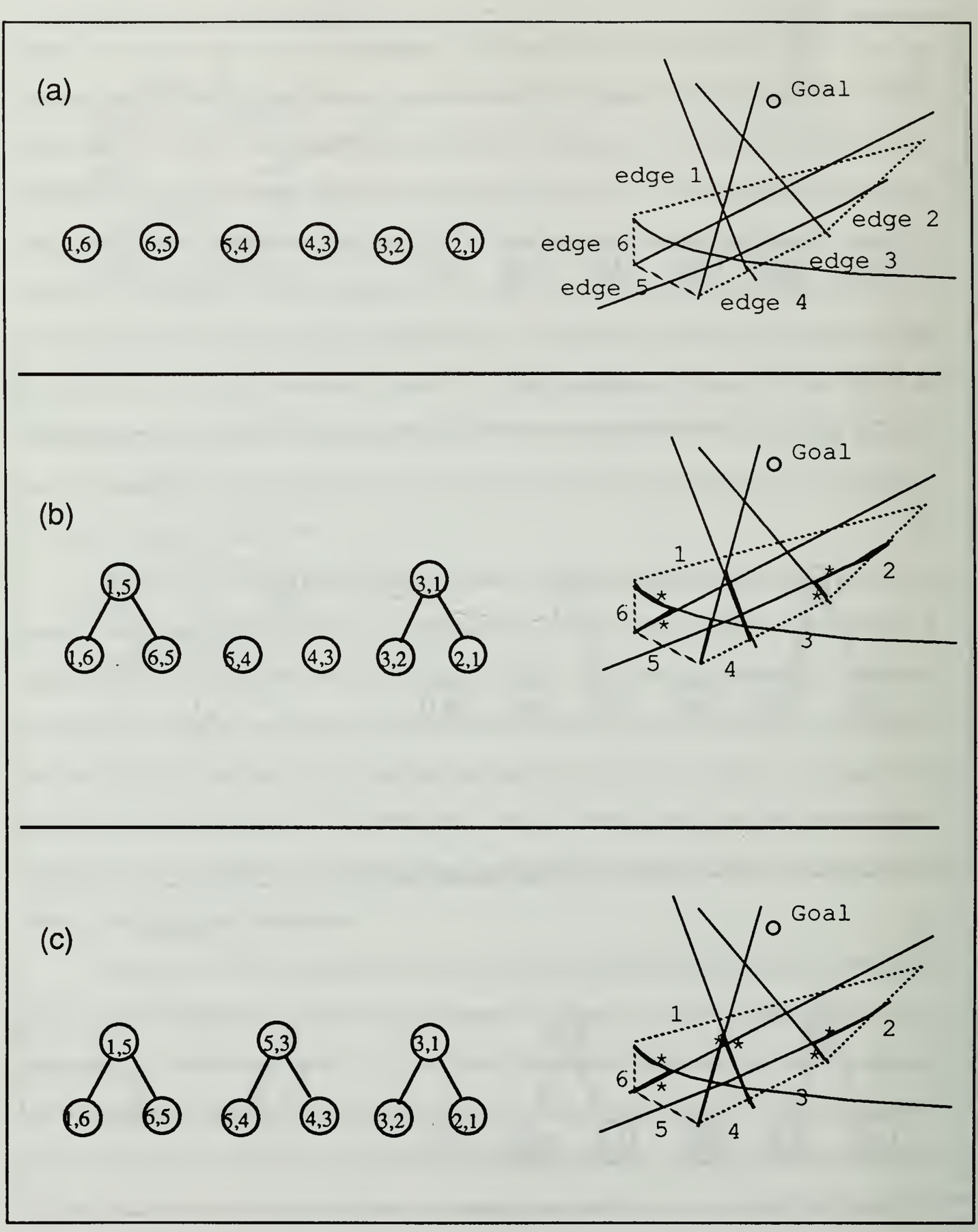


Figure 31a
Construction of HCA Interior-Boundary Tree Example 2

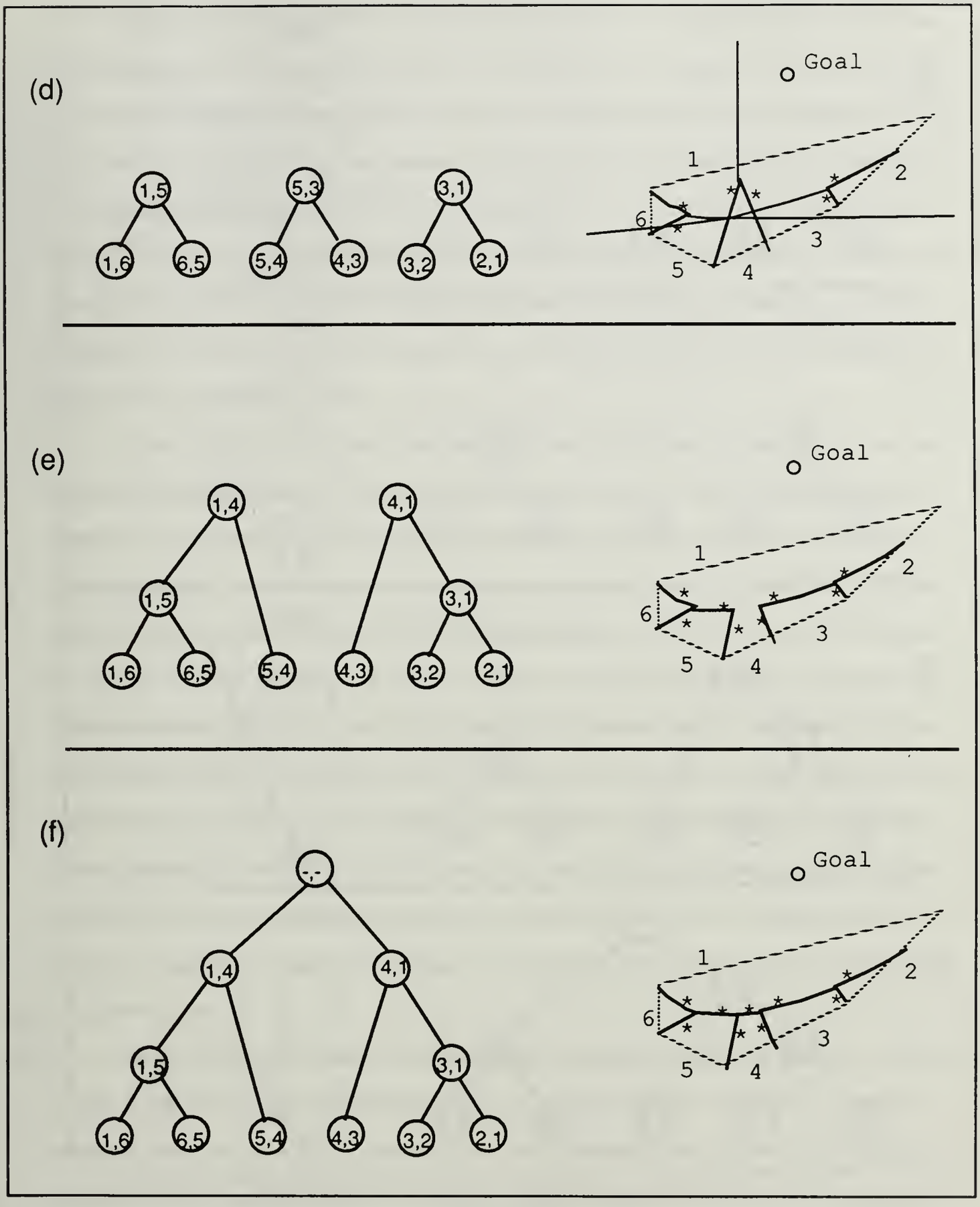


Figure 31b

Construction of HCA Interior-Boundary Tree Example 2

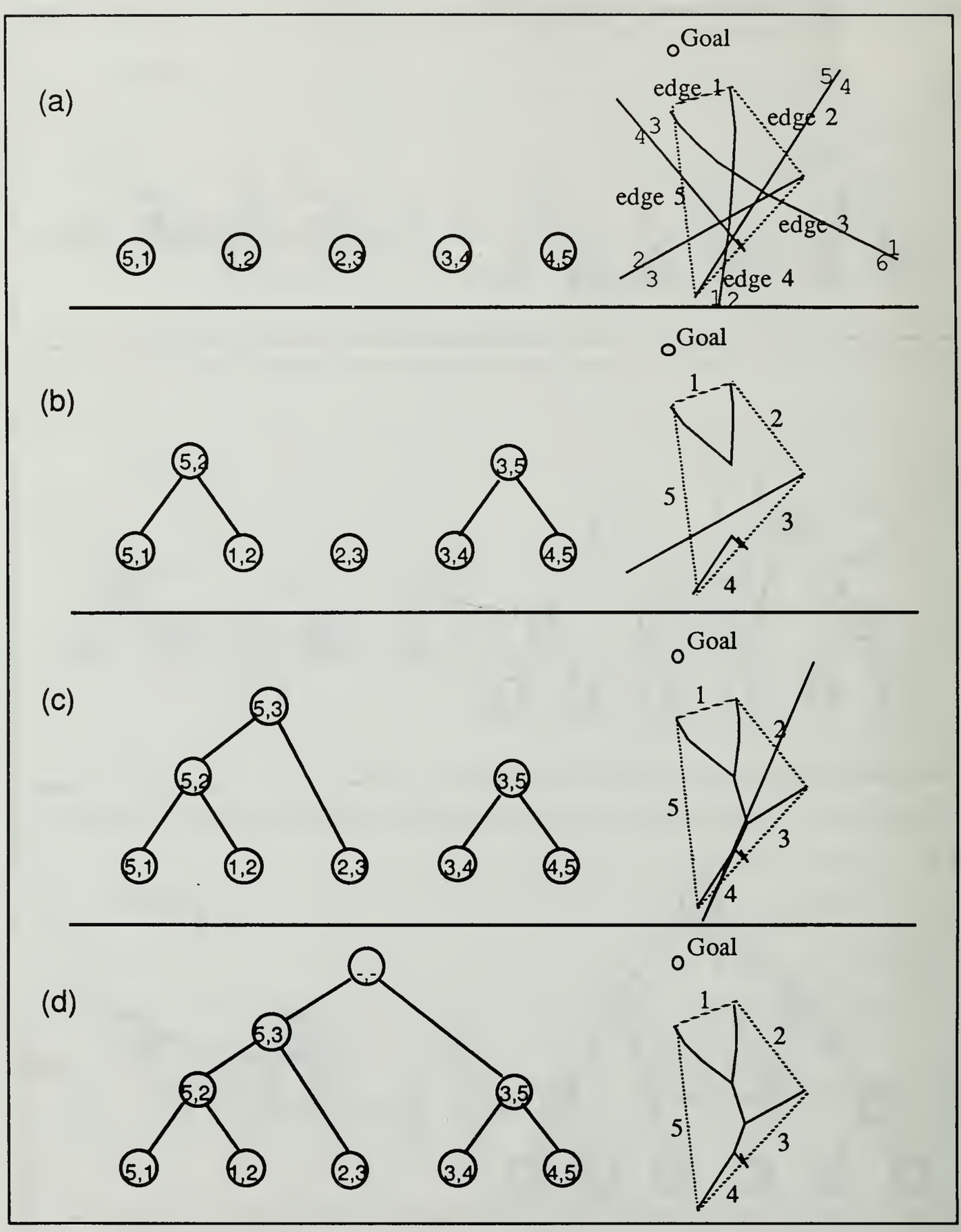


Figure 32

Construction of HCA Interior-Boundary Tree Example 3

Consider, for example, Figure 31. Figure 31b shows the state of PairedBdrySet with respect to the HCA after the first pass through the inner loop ("while PairedBdrySet is changing"), where each boundary "i,j" in the initial set of boundaries is intersected with the two adjacent boundaries, and the shortest version of "i,j" is retained in PairedBdrySet. Those boundaries which pair up with an adjacent boundary are marked with "*". In Figure 31b, "1,6" pairs with "5,6" and "1,2" pairs with "2,3". "4,5" and "3,4" were not marked, and so are going to be replaced in PairedBdrySet by the full versions of their respective boundaries at the start of the next pass through the inner loop. After the second pass through the inner loop, all boundaries are marked as in Figure 31c, so on the next iteration no changes to PairedBdrySet will be made, so the "while changing" condition will fail, ending the inner loop.

As the outer loop ("while BdrySet is changing") finishes its first pass, new boundaries are generated from each intersection point of paired boundaries, and these boundaries are placed, unmarked, into PairedBdrySet, which replaces BdrySet. This situation is reflected in Figure 31d. Figure 31e reflects the state of PairedBdrySet after the outer loop has started its second pass, and the inner loop has run until it stabilizes again. Note that some boundaries which were paired after pass one, i.e., "4,5" and "3,4", are in fact intersected by second-level boundaries instead, and so the truncated versions of the boundaries need to be retracted from PairedBdrySet and the full versions put back into PairedBdrySet for further interaction with second-level boundaries. This illustrates why such this procedure is complicated, because we are not able to tell with a single pass which boundaries will be paired. Boundary "1,4" is now propagated from both directions from the intersection points of "4,5" and "1,5" as well as "1,3" and "3,4". It is truncated at both ends and paired with itself, after which the configuration is stable. Thus BdrySet will not change further, so the outer loop will halt with BdrySet as illustrated in Figure 31f. At each stage, the interior-boundary trees are built up until, in Figure 31f, a single tree results.

In algorithm **hca-opm-high-ext**, it is assumed initially that there is an opposite point, i.e., a point on the hidden side of the HCA where two optimal paths go in opposite directions around the HCA. Further, this assumed opposite point is initially considered a vertex for the purposes of the algorithm. Figure 30 shows a situation where the algorithm leads to the conclusion that the opposite point does not exist after all, and so there is no interior boundary incident to it, because there is shortcutting of paths from the outside of the HCA across the HCA to the goal. The figure also shows a situation where there is more than one interior-boundary

tree. There is one exterior opposite-edge boundary incident upon an HCA edge associated with each interior-boundary tree. It has one endpoint at the point at which a boundary of the tree intersects an opposite edge.

5. An Algorithm for OPM Construction for A Single Convex High-Cost, Interior-Goal Homogeneous-Cost Area

A much different algorithm is needed to construct boundaries for the case of a high-cost HCA with an interior goal point (see Figure 26). The existence of interior boundaries are more predictable without the need for the iterative checking as in the high-cost, exterior-goal HCA case. It is still necessary, however, to check the intersections of various boundaries and truncate them appropriately, and insert portions of edges into the optimal-path tree, which is done at the algorithm's conclusion. (See Table 10.)

The algorithm proceeds by looking at each HCA vertex in turn, and determining by observing its optimal path whether it is a hidden or a visible vertex. If it is a hidden vertex, the path from the vertex will travel along an edge of the HCA before cutting into the interior, while if it is a visible vertex, the path will go directly to the goal. If it is hidden, several interior boundaries and one exterior shadow boundary are generated, as well as possibly an opposite-edge boundary. If it is visible, only one exterior boundary, a visible-edge boundary, is generated.

It is necessary to insert portions of edges into the optimal-path tree according to the traversal characteristics of optimal paths across or along them. For example, it is possible for a portion of an edge from one vertex to act like a road, where paths leave the HCA interior to travel along the lower-cost edge, and then cut back in to the HCA when nearer to the goal. Thus the first portion of the edge would be the root of a homogeneous-behavior region characterized by paths crossing from the interior to the exterior and travelling along the edge, and the next portion of the edge would be the root of another region characterized by paths crossing from exterior to interior. All this information is not available when processing each individual vertex, however, so edges which may become region roots are stored temporarily, and at each step when information is gained which could rule out portions of edges as roots, that information is stored as a "mask", which is used to mask out portions of edges. At the conclusion of the algorithm, these edges and masks are processed to determine exactly which portions of edges belong as region roots in the optimal-path tree. Also done at the conclusion of the algorithm is the intersecting of opposite-edge and shadow boundaries and plotting of new boundaries in the HCA exterior, much like in the interior of a high-cost, exterior-goal HCA.

TABLE 10

HIGH-COST INTERIOR-GOAL HCA OPM CONSTRUCTION ALGORITHM

```
algorithm hca-opm-high-int                                  (algorithm VI-5)
  input: Optimal-Path Tree with root N, HCA A with n vertices;
  output: DCEL M and revised OPT N;
  purpose: construct an OPM for high-cost, interior-goal HCA;
  {for i := 1 to n
    {if P ≠ Goal, where OPL(Vi) = [P | OPL(P)]          /* i.e., if path from V lies on edge E2 of HCA */
      {                                                 /* with other edge called E1, an interior linear */
      E2 := edge containing ViP;                        /* bdry and two parabolic bdrys are formed, */
      E1 := other edge incident to Vi;                  /* and an exterior shadow boundary is formed. */
      B1 := value returned by construct-high-int-hca-bdry("hidden-edge",Vi);
      B2 := value returned by construct-high-int-hca-bdry("hidden-edge/goal",Vi);
      B3 := value returned by construct-high-int-hca-bdry("visible-edge/goal",P);
      B4 := value returned by construct-high-int-hca-bdry("shadow",P);
      add B4 to ExtBdrySet;
      intersect B1, B2, B3 & add B1^trunc to IntBdrySet;  /* they intersect at the same point. */
      if B2 intersects E1 at some pt X
        {truncate B2 at X;
        add B2^trunc to IntBdrySet;
        insert-into-opt(N,ViX,"Near-side");
        add E1 and Mask(ViX) to VisEdgeSet;
        B5 := value returned by construct-high-int-hca-bdry("corner-cutting",X);
        add B5 to ExtBdrySet;
        }                                               /* if paths from two vertices go opposite ways */
      else if (OPL(Vi) ⊄ OPL(Vi+1) and OPL(Vi+1) ⊄ OPL(Vi))     /* around HCA, edge is opp edge. */
        {B6 := value returned by construct-high-int-hca-bdry("interior-opposite-edge",Vi,Vi+1);
        intersect B6 with B2 & add B6^trunc to IntBdrySet;
        X := pt where B6 intersects E1;
        B7 := value returned by construct-high-int-hca-bdry("exterior-opposite-edge",Vi,Vi+1);
        add B7 to ExtBdrySet;
        insert-into-opt(N,ViX,"Near-side");
        insert-into-opt(N,XVi+1,"Near-side");
        }
      else
        {insert-into-opt(N,E1,"Near-side");
        add B2^trunc to IntBdrySet;
        }
      if B3 intersects edge E2 at X
        {insert-into-opt(N,ViX,"Near-side");
        truncate B3 at X;
        add B3^trunc to IntBdrySet;
        }
      }
    else                                                /* i.e., if path goes from V directly to Goal. */
      {B := value returned by construct-high-int-hca-bdry("visible-edge",Vi);
      add B to ExtBdrySet;
      add E1 and E2 to VisBdrySet;
      }
  post-process-high-int-hca-bdrys;
  }                                                     /* end of hca-opm-high-int */
```

TABLE 10 (CONTINUED)

HIGH-COST INTERIOR-GOAL HCA OPM CONSTRUCTION ALGORITHM

```
procedure construct-high-int-hca-bdry                    /* constructs each type of bdry formed by*/
   input: type of bdry T; P, the start-point of bdry;    /* a high-cost, interior-goal HCA. */
   output: Bdry, the resulting boundary;
   purpose: to construct a boundary generated from point P of type T;
   {if T = "hidden-edge" Bdry := curve as specified in Lemma V-5.1;
   if T = "hidden-edge/goal" Bdry := curve as specified in Lemma V-5.2;
   if T = "visible-edge/goal" Bdry := curve as specified in Lemma V-5.3;
   if T = "interior-opposite-edge" Bdry := curve as specified in Lemma V-5.4;
   if T = "hca-edge" Bdry := curve as specified in Lemma V-5.5;
   if T = "shadow" Bdry := curve as specified in Lemma V-5.6;
   if T = "exterior-opposite-edge" Bdry := curve as specified in Lemma V-5.7;
   if T = "corner-cutting" Bdry := curve as specified in Lemma V-5.8;
   if T = "visible-edge" Bdry := curve as specified in Lemma V-5.9;
   }                                                     /* end of construct-high-int-hca-bdry */

procedure post-process-high-int-hca-bdrys                /* store bdrys and edges. */
   input: VisEdgeSet, the set of bdrys from visible edges, IntBdrySet, the set of
      interior bdrys, ExtBdrySet, the set of exterior bdrys, and Optimal-Path Tree N;
   output: DCEL M, and revised OPT N;
   {for each edge E ∈ VisEdgeSet
      {for all MaskE E := E less MaskE;
      N := value returned by insert-into-opt(N,E);
      }
   for each bdry B ∈ IntBdrySet
      if another version of B exists
         {truncate B and B′;
         insert-into-dcel(M,Btrunc);
         }
   for each bdry B ∈ ExtBdrySet
      join-high-int-bdrys(B,ExtBdrySet);
   for each bdry B ∈ ExtBdrySet
      insert-into-dcel(M,B);
   }                                                     /* end of post-process-high-int-hca-bdrys */

procedure join-high-int-bdrys                            /* joins external bdrys. */
   input: bdry B1, set of bdrys ExtBdrySet;
   output: revised ExtBdrySet;
   purpose: to pair bdrys which first intersect, and propagate new ones from their pt of intersection.
   {for each B2 ∈ ExtBdrySet such that B1 and B2 intersect and B1 and B2 are adjacent
      {truncate B1 and B2;
      remove original B1 and B2 from ExtBdrySet;
      add B1trunc and B2trunc to ExtBdrySet;
      T := type of new bdry;                             /* based on edges E1 and E2 not */
      B3 := value returned by                            /* common to B1trunc and B2trunc. */
      construct-hca-opm-high-int-bdry(T,E1,E2)
      join-high-int-bdrys(B3,ExtBdrySet);                /* recursively follow bdry outward from HCA. */
      }
   }                                                     /* end of join-high-int-bdrys. */
```

6. An Algorithm for OPM Construction for A Single Low-Cost, Exterior-Goal Homogeneous-Cost Area

The exterior-goal-low-cost-region algorithm shown in Table 11 looks at each HCA vertex in turn, basing its logic on the initial direction of the optimal path from the vertex being examined (see Figure 27). If the optimal path from a vertex goes into the HCA interior, two rays, or vertex/edge-crossing boundaries, are constructed forming a wedge outward from the vertex and away from the goal. If the optimal path goes along an edge of the HCA, one of the above boundaries, the one closer to the direction of travel of the optimal path, is instead a vertex/edge-following boundary, and in addition a parabolic, or vertex/goal boundary is constructed. The third possibility is that the optimal path goes directly into the HCA exterior, i.e., toward the goal. If so, more boundaries may or may not be generated. If a portion of each edge adjacent to the vertex is visible to the goal, i.e., if for both edges there are paths starting at some points on the edges which go directly into the HCA exterior, then a visible-edge boundary will emanate from the vertex into the HCA interior.

With the above boundaries generated, two tasks remain. First, each parabolic, or edge-following/goal boundary must be followed away from the goal to see if it intersects the next ray boundary. If so, a hyperbolic, or vertex/goal boundary will begin, with one focus at the vertex. This hyperbola must then be followed in turn. If it intersects a ray boundary, a "distorted-parabolic", or edge-crossing/goal boundary will begin. As we continue to follow this sequence of boundaries, hyperbolas and distorted-parabolas occur alternately until no intersection with a ray is found. Note that this algorithm generates each parabolic and distorted-parabolic boundary in the initial phase, and then generates hyperbolas as needed in procedure **add-hyp-bdrys-for-low-ext-hca** below, which in addition truncates each boundary as necessary.

Although this type of HCA has interior boundaries, which one might suppose would need to be paired and merged as with the high-cost, exterior-goal case, in fact it is not necessary to do this. The reason is that such boundaries are all of the visible-edge type, and because the HCA interior is of lower cost than the surrounding terrain, these boundaries will never intersect. Intuitively in the high-cost exterior-goal case, a path travels to an edge further away in straight-line distance in order to take advantage of the lower external cost outside that edge, and at that point, two boundaries would intersect and a third emerge. Here, however, the path is already in the least costly terrain possible, and so further paths will continue to follow the same paths as those closer to the goal. For each visible-edge boundary, a point of intersection is plotted with the far edge

TABLE 11

LOW-COST EXTERIOR-GOAL HCA OPM CONSTRUCTION ALGORITHM

```
algorithm hca-opm-low-ext                                   (algorithm VI-6)
  input: Optimal-Path Tree with root N, and HCA A;
  output: Optimal-Path Map M  (a DCEL) and modified Optimal-Path Tree N;
  purpose: construct an OPM for a low-cost, exterior-goal HCA;
  {
  for each vertex V of A with incident edges E1 and E2   /* consider each vertex and its adjacent edges */
    such that E1 = VV1 and E2 = VV2, where |V2G| ≥       /* where V1 is closer to goal than V2, and */
    |V1G| and OPL(V) = [P | OPL(P)]                      /* where P is the first point on V's opt path. */
    {
    if VP lies in HCA interior                          /* if optimal path from V goes into HCA interior */
      {
      construct-low-ext-hca-bdry(vertex/
        edge-crossing,V,V1,V2);                         /* two rays are Snell's-Law paths across */
      construct-low-ext-hca-bdry(vertex/                /* edges E1 and E2 through vertex V */
        edge-crossing,V,V2,V1);
      insert-into-opt(N,VV1,Far-side);                  /* add edges to OPT as region roots. */
      insert-into-opt(N,VV2,Far-side);
      }
    else if VP lies along HCA edge E1                   /* if opt. path from V goes along an HCA edge */
      {
      construct-low-ext-hca-bdry(edge-following/
        goal,V,V1,V2);
      construct-low-ext-hca-bdry(vertex/
        edge-following,V,V1,V2);                        /* two rays are Snell's-Law paths across */
      construct-low-ext-hca-bdry(vertex/                /* edges E1 and E2 through vertex V */
        edge-crossing,V,V2,V1);
      insert-into-opt(N,VP,Near-side);                  /* add edges to OPT as region roots. */
      insert-into-opt(N,VV2 Far-side);
      }
    else if ((Q1 is in HCA exterior) or (V1Q1∈ V1V))    /* if both edges are visible or partially visible */
      and ((Q2 is in HCA exterior) or (V2Q2∈ V2V)))     /* (optimal path from V lies in HCA exterior). */
      where OPL(V1) = [Q1 | OPL(Q1)]                    /* Note: Qi are the first points on */
      and OPL(V2) = [Q2 | OPL(Q2)]                      /* the optimal-path lists of each Vi */
      {
      construct-low-ext-hca-bdry(visible-edge,          /* vis-edge bdry from V w.r.t. E1 and E2 */
        V,V1,V2);
      for i := 1 to 2
        if (V1Q1∈ V1V)                                  /* add as region root the portion of */
          insert-into-opt(N, Q1V, Far-side);            /* edge across which paths cross. */
        else insert-into-opt(N,  V1V, Far-side);
      }
    }
    add-hyp-bdrys-for-low-ext-hca(ParabBdrys);
    }                                                   /* end of algorithm hca-opm-low-ext */
```

TABLE 11 (CONTINUED)

LOW-COST EXTERIOR-GOAL HCA OPM CONSTRUCTION ALGORITHM

```
procedure add-hyp-bdrys-for-low-ext-hca;          /* puts hyperbolas between pairs of ray bdrys */
  input: ParabBdrys, the set of parabolic boundaries;
  output: revised DCEL;
  purpose: to concatenate hyperbolic bdrys onto parabolic ones.
  {
  while ParabBdrys ≠ ∅
    {
    select bdry B1∈ ParabBdrys associated with vertex Vi and edge Ej ;
    truncate B1 and the vertex/edge-following or vertex/edge-crossing bdry B2 emanating
      from Vi, and associated with edge Ej at the point where they intersect;
    B4 := value returned by construct-low-          /* bdry is hyperbola intersecting one or */
      ext-hca-bdry(vertex/goal,Vi,Vj,Vk);           /* both rays emanating from vertex Vi */
    if B4 intersects vertex/edge-crossing bdry
      B3 associated with Vi and edge Ek, k≠j,
      {
      truncate B3 and B4 at their point of intersection;
      truncate B5∈ ParabBdrys assoc. with Vg and Ek, g≠i, at its intersection with B3 and B4 ;
      }
    remove B1 from ParabBdrys;
    }                                               /* end add-hyp-bdrys-for-low-ext-hca */
```

of the HCA, and an opposite-edge boundary is generated, which is really just a continuation of the visible-edge boundary after crossing another edge.

Procedure **construct-low-ext-hca-bdry** performs the low-level function of generating each boundary for the low-cost, exterior-goal HCA as needed. For boundaries whose forms are general curves, the reader is referred to the appropriate Lemma in Chapter V and proof in Appendix A.

7. An Algorithm for OPM Construction for A Single Low-Cost, Interior-Goal Homogeneous-Cost Area

Algorithm **hca-opm-low-int** is the simplest of the four HCA algorithms, in keeping with the simple nature of the regions and boundaries associated with this type of HCA (see Figure 28 and Table 12). Since a low-cost, interior-goal HCA generates only one wedge of two rays at each vertex, and these rays are guaranteed by the orientation of the HCA edges not to interact, the corresponding algorithm can do its work in one pass through the list of vertices.

TABLE 12

LOW-COST INTERIOR-GOAL HCA OPM CONSTRUCTION ALGORITHM

```
algorithm hca-opm-low-int                                   (algorithm VI-7)
  input: Optimal-Path Tree with root N, and HCA A;
  output: Optimal-Path Map M  (a DCEL) and modified Optimal-Path Tree N;
  purpose: to construct the OPM for a low-cost,, interior-goal HCA;
  {
  M := empty dcel structure;
  for each edge V1V2 of A
     {
     Bdry1 := ray starting at V1, lying away                /* two bdrys emanate from each vertex, */
        from Goal G thru pt X1, such that                   /* at the Snell's-Law angle with respect */
        ∠GV1V2= π/2−θ1, ∠X1V1V2 = π/2+θ2 ,                  /* to each edge. */
        and cint sinθ1 = cext sin θ2;
     Bdry2 := ray starting at V2, lying away
        from Goal G thru pt X2, such that
        ∠GV2V1= π/2−θ1, ∠X2V2V1 = π/2+θ2 ,
        and cint sinθ1 = cext sin θ2 ;
     insert-into-dcel(M,Bdry1);                             /* add vertex/edge-crossing bdrys to DCEL.*/
     insert-into-dcel(M,Bdry2);
     insert-into-dcel(M,V1V2);                              /* add HCA-edge boundary to DCEL */
     insert-into-opt(N,edge V1V2,Far-side);                 /* add edge which is crossed to OPT. */
     }
  }                                                         /* end of hca-opm-low-int */
```

C. EXTENDING THE BASIC ALGORITHMS TO MULTIPLE CONNECTED RIVER AND ROAD SEGMENTS

1. An Algorithm for OPM Construction for Multiple Connected River Segments

It is now possible to build on a basic understanding of the nature of boundaries generated by single, isolated river segments in order to construct the boundaries associated with multiple, connected linear river segments, or *rivers*. There may be two or more river segments emanating from a single vertex, but all segments of a river must have the same crossing cost. It might be thought that the algorithm proposed below to construct the optimal-path map for multiple terrain features could be used to construct it for this kind of terrain as well. However, connected river segments are not "decomposable" into their constituent segments. *Decomposability* of a set of terrain features is defined as follows.

Terrain map M with optimal-path tree N is defined as *decomposable for path planning* into subsets S_1 and S_2 if both S_1 and S_2 are consistent with OPT N. Say that a set of terrain S_i which is a subset of a set of terrain S is *consistent* with an OPT N constructed for S if for an OPT N_i constructed for S_i considered alone, every node of N_i appears in N, and the parent of every node of N_i appears in the path from the node to the root of N.

In other words, if one subset could not behave in the way it does without the presence of the other, the terrain is not decomposable. Connected river segments have as part of their nature that at internal vertices, i.e., where two segments join, there are regions where paths must either cross a river or move away from the vertex, while for the individual segments, a path could bypass the river segment by simply moving around the vertex. Thus a set of connected river segments is not decomposable into its individual segments.

Two high-level paradigms in addition to those used for single river segments are useful here. First, we partially sort the river segments according to their general visibility to the goal, i.e., so that a segment which is fully or partially occluded by another follows it in the partial order, and we process the segments according to this partial order. Thus boundaries which may affect other segments are already in place by the time the other segments are considered. Second, whenever a boundary intersects an occluded segment, an *event point* is generated. When a segment is processed, it is necessary to consider each event point and decide whether the boundary which caused the event point continues on the other side of the river segment. Figure 33 shows a river consisting of connected river segments, and Figure 34 shows a worst-case orientation of segments.

Several new terms must be defined. *General visibility* between two terrain features is defined as follows. Two features F_1 and F_n are generally visible with respect to a goal G if there is a sequence of features F_i, i=1 to n, such that for all i, F_i is visible to F_{i+1}. A feature F_i is *occluded* by another F_j with respect to goal G if for every sequence by which F_i and G are generally visible, F_j is a member of the sequence. In other words, F_i is occluded by F_j if it is partially or completely within the shadow of F_j cast by G. An endpoint V of line segment L_1 is defined as an *exterior vertex* if V is not an endpoint of any other line segment, or if segment L_2 of which V is an endpoint occludes L_1. V is defined as an *interior vertex* if it is not an exterior vertex. Intuitively this means that an optimal path from an interior vertex must either cross the river or move away from the vertex to get past the line segment, while from an exterior vertex an optimal path can simply move around the vertex and bypass the river. Figure 33 shows the partial ordering of river segments as well as the exterior or interior nature of each vertex.

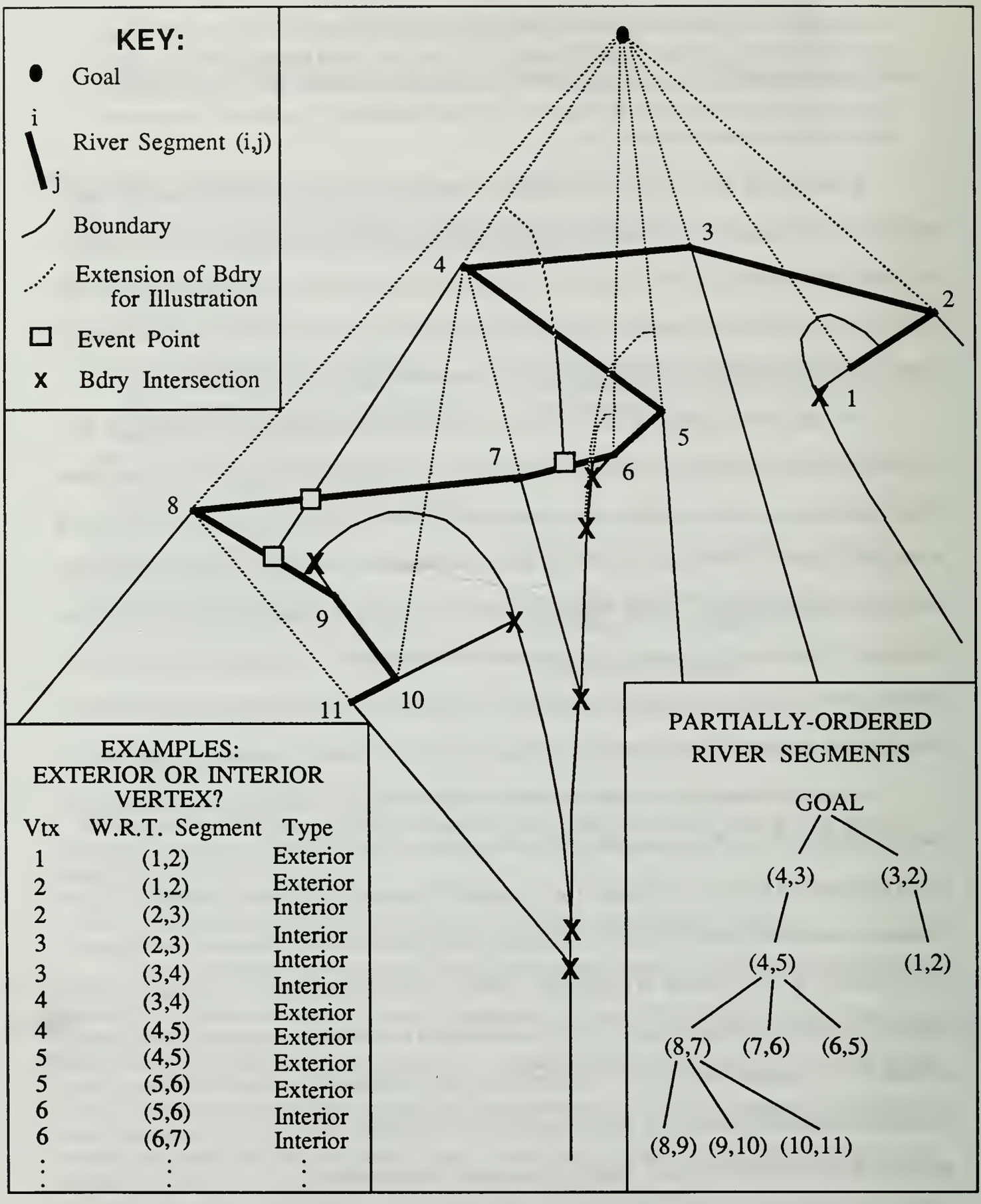


Figure 33

Multiple Connected River Segments

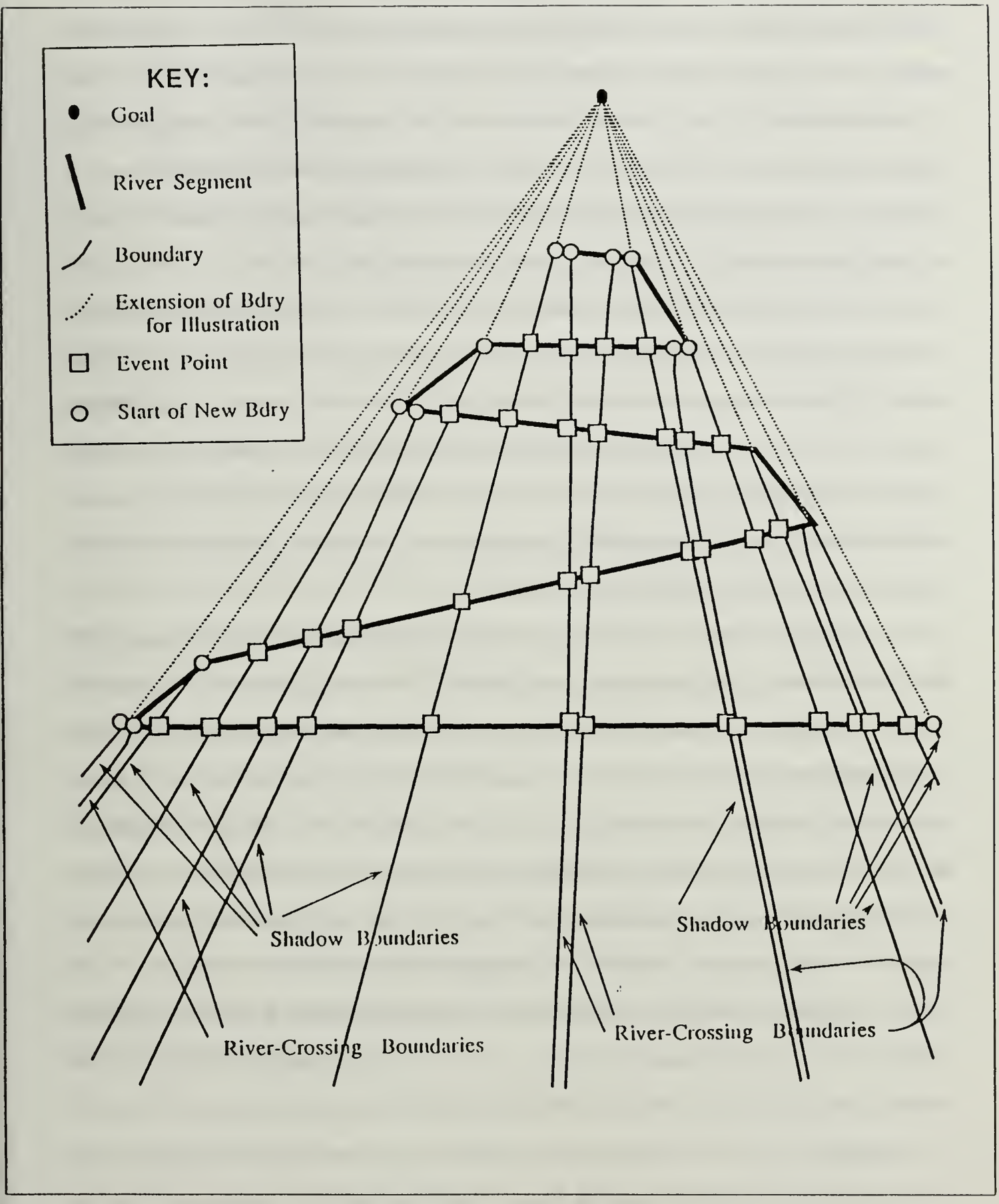


Figure 34

Worst-Case Complexity of Multiple Connected River Segments

The algorithm based on these ideas is complicated by the possibility that rivers may turn back on themselves and create pockets where, for a high-enough crossing cost, it is cost-effective to move away from the goal out of the pockets rather than cross a river. This situation is illustrated in Figure 33a, in the vicinity of vertices 8 through 11. A type of boundary in addition to those presented in Chapter V for single river segments is generated in this case, although it is very similar to the other types. Whenever an exterior vertex V_1 is encountered in the course of processing river segments, a river-crossing boundary is generated for that vertex, as explained in Chapter V. If this boundary does not intersect any segment between V_1 and V_2, all paths from immediately on the far side of the river including the path from V_2 will go via V_1. In this situation there will be a boundary which separates paths which cross a river toward the goal from those which move away from the goal and eventually go through V_2 and then through V_1. At each interior vertex, as well as at the next exterior vertex, a portion of this boundary will be generated. This type of boundary is called a *near-side-river-crossing boundary* and it is exactly the complement of the river-crossing boundary which would be generated from that vertex if an optimal path from the vertex lay forward across the river. In other words, it starts at the current river segment and lies forward toward the goal.

An example of a near-side-river-crossing boundary in Figure 33 has one end-point on segment (8,9). From there, it lies toward the goal until it intersects a shadow boundary which starts at vertex 9. The next portion of the boundary is the hyperbola segment whose axis is the line between vertices 10 and 4. After it intersects the shadow boundary from vertex 10, the boundary is the hyperbola segment whose axis is the line between vertices 11 and 4. Finally, it ends at the point where it intersects a river-crossing boundary separating points whose paths go around vertex 5 from those whose paths cross segment (7,8). From there an opposite-edge boundary begins, separating points whose paths go around vertex 5 from those which go around vertex 11. A second example of a near-side-river- crossing boundary is in the vicinity of vertex 1.

At each interior vertex V, the test for a near-side-river-crossing boundary is as follows. If for a point arbitrarily close to V, but on the near side of the river, called V^-, the optimal-path list of V^- includes V_{ext}, the currently active exterior vertex, a near-side-river-crossing boundary is generated. The foci are V and the vertex or goal point X such that the cost of a straight-line path from V to X plus the cost of the optimal path from X is minimized, and the hyperbolic constant is the cost of the optimal path from V minus the sum of the cost of the rivers crossed from V to X and the cost of the optimal path from X. In this case, the shadow boundary

from V may intersect the boundary at a point P. The boundary starts at its intersection with the current river segment, or if it is the second or subsequent portion of the boundary to be generated, at its intersection with the previous boundary portion, and ends at point P. If the boundary intersects the river-crossing boundary generated by V_{ext}, it ends at that point and the V_{ext} boundary constitutes the remainder of the boundary.

Shadow boundaries follow the same specifications as listed in Chapter V, namely that a vertex V, with an optimal path which goes first to point P, generates a shadow boundary which is a ray starting at V and lying away from P on the line VP, but with the variation that it must be considered whether the optimal path of V crosses the river segment or not. If for V^- as defined above, and for V^+ arbitrarily close to V on the far side of the river segment, V^- can be positioned so as to lie on the optimal path of V^+, a normal shadow boundary results. This is the case where paths from the far side of the river may cross in the vicinity of V, and such a boundary simply keeps track of which segment the paths cross.

If V^- includes in its optimal path the current V_{ext}, it will be the case that V^+ does so as well, and the shadow boundary which results will conform to the above specification with respect to V^- and will have the unusual characteristic that it lies on the near side of the river segment. This is the case where points on the near side of V are caught in a "pocket" for which it is faster to move away from the goal and around V_{ext} than to cross river segments forward of V. Such a shadow boundary separates points which go to V and then to the vertex on the next river segment from those which bypass V and go to the next vertex directly.

If V^+ includes in its optimal path the current V_{ext}, but V^- does not, the shadow boundary from V will be formed with respect to V^+, and lie on the far side of the river. This is the case where the optimal path of V^- lies on the same side of the river segment as does the optimal path of V_{ext}, but the optimal path of V^+ includes V_{ext}, signifying that paths on the far side of the river in the vicinity of V will not cross it, but paths on the near side will lie generally toward the goal, not being caught in a "pocket" which causes them to move away from the goal to avoid crossing subsequent river segments. Table 13 shows the algorithm for construction of multiple connected river segments.

TABLE 13

MULTIPLE-CONNECTED-RIVER-SEGMENT OPM CONSTRUCTION ALGORITHM

```
algorithm multi-segment-river-opm                    (algorithm VI-8)
  input: List of river segments R, river-crossing cost Cr, Optimal-Path Tree N, Goal G;
  output: DCEL M and revised OPT N;
  purpose: construct a planar partition and revise the OPT for multiple, connected river segments;
  {partially order R so that Si < Sj iff Si obscures part of Sj with respect to G;
  for each segment S ∈ R in partial order, letting S = V1V2 where V1 is closer to G than is V2
    {plot shadow bdry from V1;
    plot shadow bdry from V2;
    intersect bdrys with all subsequent segments, noting an event-point whenever intersection occurs;
    if V1 is "exterior"
      {Vext := V1;
      sort event-pts on V1V2 with respect to V1,
        including V2 as an event-point;
      until a river-crossing bdry is plotted for V1
        or event-list is empty
        {select next event-point E;
        Er := root of region on side of E closer to V1;
        CVext := |EVext| + |(VextG)*|;
        CEr := |EEr| + |(ErG)*|;
        if CVext > CEr
          plot river-crossing bdry with respect to Vext and Er;
        else
          delete portion of bdry BE lying away from Goal;
        }
      }
    else
      do nothing;                                    /* if V1 is "interior". */
    if V2 is "interior"
      {V2- := point arbitrarily close to V2 on same side of river
        as first leg of optimal path from Vext;
      V2+ := point arbitrarily close to V2 on opposite side of river
        as first leg of optimal path from Vext;
      if Vext ∈ (V2-G)* and Vext ∈ (V2+G)*
        {plot shadow bdry B1 such that for OPL(V2-G)* = [P | OPL(P)],
          B1 := ray on line V2P starting at V2, lying away from P;
        X := vertex or goal such that                /* ie, vertex with best cost from V2 to X to Goal, */
          |V2X| + |(XG)*| is minimized;              /* with hyp cost := Cr·no. rivers crossed by V2X. */
        plot B2 := near-side-crossing bdry with      /* note that if intersection is beyond V1, there is */
          foci V2 and X, where B2 starts at point of /* another near-side-crossing bdry which */
          intersection with line V1V2 and ends at    /* intersects B2. */
          intersection with B1;
        }
      else if P- = P+, where OPL(V2-) = [P- | OPL(P-)] and OPL(V1+) = [P+ | OPL(P+)]
        plot shadow bdry on line V2P+ starting at V2 lying away from P+;
      }
```

TABLE 13 (CONTINUED)

MULTIPLE-CONNECTED-RIVER-SEGMENT OPM CONSTRUCTION ALGORITHM

```
    else                                          /* V2 is "exterior". */
      if Vext ∈ (V1⁻G)* and Vext ∈ (V1⁺G)*        /* if V2 is "hidden". */
        {plot shadow bdry B1 such that for OPL(V1⁻G)* = [P | OPL(P)],
          B1 := ray on line V1P starting at V1, lying away from P;
        X := vertex or goal such that             /* ie, vertex with best cost from V1 to X to Goal, */
          |V1X| + |(XG)*| is minimized;           /* with hyp cost := Cr·no. rivers crossed by V2X. */
        plot B2 := near-side-crossing bdry with   /* note that if intersection is beyond V2, there is */
          foci V1 and X, where B2 starts at point of  /* another near-side-crossing bdry which */
          intersection with line V1V2 and ends at     /* will intersect B2. */
          intersection with B1;
        }
      else                                        /* V2 is "visible". */
        {Vext := V2;
        sort event-pts on V1V2 with respect to V2,
          including V1 as an event-point;
        until a river-crossing bdry is plotted for V2
          or event-list is empty
          {select next event-point E;
          Er := root of region on side of E closer to V2;
          CVext := |EVext| + |(VextG)*|;
          CEr := |EEr| + |(ErG)*|;
          if CVext > CEr
            plot river-crossing bdry with respect to Vext and Er;
          else
            delete portion of bdry BE lying away from Goal;
        }
    join and merge bdrys associated with V1V2, noting all
      intersections with obscured segments as event-points.
    }                                             /* end of "for each segment". */
  join and merge all bdrys;
  }                                               /* end of algorithm multi-segment-river-opm. */
```

2. OPM Construction of Multiple Connected Road Segments

Unlike connected river segments, connected road segments are decomposable into their constituent segments. The basic reason for this is that road segments will not serve to block or hinder paths, but only to operate as conduits. Therefore, connected road segments can be decomposed into individual segments by algorithm VI-9 below, algorithm VI-3 used on each segment, and the resulting OPM's merged into a final OPM.

D. A DIVIDE-AND-CONQUER ALGORITHM FOR MULTIPLE-FEATURE OPM CONSTRUCTION

A principal goal of our research is to find an algorithm which will create optimal-path maps for multiple terrain features of the four types described above. Although the investigation into this problem is not complete in all its details, we propose the following high-level description of such an algorithm (see Table 14).

Methods for constructing Voronoi diagrams (see Chapter II) provide a model for approaches to the construction of an optimal-path map for multiple terrain features. Voronoi diagram methods use a divide-and-conquer approach, in which the points in the plane are divided into two roughly equal sets, the Voronoi diagrams of the two sets computed recursively, and the two Voronoi diagrams merged to produce the final one. The first key question is how to divide the points in the plane. The answer in this case is that in order to support the merge phase, the plane is partitioned into two half-planes by a line (by convention, a vertical line) which equally divides the set of points in the plane. The other key question is whether the two intermediate Voronoi diagrams can be merged. Standard generalized-Voronoi-diagram construction algorithms provide an affirmative answer to this question, depending on the fact that the boundary between any two Voronoi regions in binary terrain (i.e., obstacles on a homogeneous-cost background) is a straight line segment or a hyperbola segment [Ref. 8].

The analogous questions with respect to optimal-path map construction are whether terrain features can be divided in the same manner as points, and how two optimal-path maps with the same goal can be merged into a single, combined OPM. An encouraging aspect of this problem is that when constructing OPM's for single terrain features, we rely on the optimal paths from only the terrain-feature vertices, which are computed by standard point-to-point path planners and take all the features of a map into account. Thus the optimal paths from any vertex will remain the same regardless of which terrain features are incorporated into the OPM. Another important aspect of this problem is the unifying perspective with regard to regions and boundaries proposed in Chapter V, Section C. Since there are only three types of non-degenerate region roots, i.e., points, edges traversed length-wise, and edges traversed cross-wise (according to Snell's Law), it should be possible at the intersection of any two general boundaries to generate a new boundary by considering the six types of boundaries between regions of three possible types of roots. Actually, as discussed in Chapter V, the Snell's-Law edges do not comprise a single class of region roots, because edges with different numbers of edge-cross-

TABLE 14

MULTIPLE-FEATURE OPM CONSTRUCTION ALGORITHM

```
algorithm multiple-feature-opm                    (algorithm VI-9)
   input: L a listing of lists of vertices and types of each terrain feature, and
      N, the optimal-path tree associated with Map;
   output: M, a DCEL describing the planar partition of OPM associated with Map, revised OPT N;
   purpose: to construct an optimal-path map for input map consisting of any number of terrain features;
   {
   if Map contains only one terrain feature                /* base case of the recursion. */
      OPM := appropriate single-feature algorithm;
   else
      {
      (Set1,Set2) := value returned by halve-map;          /* divide map into two roughly equal sub-maps. */
      N1 := N less region roots associated with Set2;
      N2 := N less region roots associated with Set1;
      OPM1 := value returned by multiple-feature-opm(Set1, N1);/* recursively solve each sub-problem. */
      OPM2 := value returned by multiple-feature-opm(Set2, N1);
      OPM := result of merge-opms(OPM1,OPM2,N1,N2).       /* find OPM by merging two sub-OPM's. */
      }
   }                                                      /* end of multiple-feature-opm. */


procedure halve-map
   input: Map, the list of lists of terrain-feature vertices and types;
   output: a pair of sets such that the first is the left half of the map and the second is the right half;
   purpose: divide Map into two roughly equal-sized sub-maps;
   {
   for each decomposable terrain feature
      find the left-most vertex;
   compute the median x-coordinate;
   for each terrain feature F
      if its left-most vertex is less than the median
         Set1 := Set1∪{F};
      else
         Set2 := Set2∪{F};
   }                                                      /* end of halve-map. */
```

TABLE 14 (CONTINUED)

MULTIPLE-FEATURE OPM CONSTRUCTION ALGORITHM

```
procedure merge-opms
   input: OPM1 and OPM2, DCEL's of the two input OPM's;
   output: OPM, the DCEL containing the planar partition of the merged map;
      OPT N, the revised optimal-path tree;
   purpose: merge two OPM's into one;
   {σ := vertical chain such that all terrain features of OPM1 are entirely
      to its left and all terrain features of OPM2 are to its right;
   BdrySet := Set of all B such that B∈ OPM1 or B∈ OPM2 and B intersects σ;
   while BdrySet is changing
      {PairedBdrySet := BdrySet;
      while PairedBdrySet is changing
         {for all Bi,j∈ PairedBdrySet                    /* where h, i, j, and k index the regions of OPM. */
            where Bi,j is unmarked
            {discard Bi,j from PairedBdrySet;
            add Bi,j from BdrySet to PairedBdrySet;
            intersect Bi,j with Bh,i and truncate both;
            add Bh,i^trunc and Bi,j^trunc to PairedBdrySet;
            intersect Bi,j with Bj,k andtruncate both;
            add Bi,j^trunc and Bj,k^trunc := to PairedBdrySet;
            }
         for all Bi,j∈ PairedBdrySet
            discard all but the shortest Bi,j from PairedBdrySet;
         unmark all bdrys in PairedBdrySet;
         for all Bi,j and Bj,k∈ PairedBdrySet such that Bi,j adjoins Bj,k
            mark Bi,j and Bj,k;
         }
      for all Bi,jand Bj,k∈ PairedBdrySet such that Bi,j adjoins Bj,k
         add Bi,k to PairedBdrySet;
      BdrySet := PairedBdrySet;
      }
      for each new B′ ∈ BdrySet
         {intersect-and-merge(B′, {all bdrys from OPM1 assoc with R1});
         intersect-and-merge(B′, {all bdrys from OPM2 assoc with R2});
         }
   }                                                   /* end of merge-opms. */

procedure intersect-and-merge
   input: B, a new boundary, and BdrySet, a set bdrys potentially intersecting B;
   output: revised DCEL M;
   purpose: propagate the effects of new boundary B in one of the subordinate OPM's;
   {for each bdry B′ ∈ BdrySet
      if B intersects B′
         {truncate B and B′at their point of intersection;
         find regions Ri and Rj which are adjacent to B and B′respectively, but not common to both;
         construct Bnew by referring to the roots of Ri and Rj respectively;
         for each boundary B″in BdrySet which B′previously intersected
            intersect-and-merge(B″, BdrySet less B′);
         }
   }                                                   /* end of intersect-and-merge. */
```

ing episodes enroute to the goal will create boundaries of different analytic characteristics. However, the concept is promising.

To divide decomposable terrain features (see Section C above for a definition of decomposability) of the input map into two approximately equal sets whose resulting OPM's can be merged is not difficult. In fact, it appears that any partition is feasible as long as it does not split a terrain feature, but some will be much more efficient than others. Of course the advantage of a divide-and-conquer algorithm is its logarithmic performance in the recursive stage if it is guaranteed that divisions are approximately equal-sized, so any partitioning procedure should have this property. Also, it should not take an excessive amount of time to accomplish the partition, since this step will play an important part in the overall time complexity. And thirdly, since the merging step will depend on checking for intersections between all boundaries of one sub-OPM and all boundaries of the other, it would be very useful if it were not necessary actually to check most of these boundaries. This would be the case if at each step in the recursion, the two OPM's represented terrain which did not, loosely speaking, "interleave". For such OPM's, boundaries which lay wholly within the interior of the two planar partitions would not have to be checked for intersection.

The merge step depends on the fact that any two boundaries, when they intersect, represent the meeting point of three regions, one of which is common to both boundaries. A new boundary will emanate from the point of intersection which separates the two regions which the original boundaries did not have in common. Rather than attempt to study all the special cases of possible region intersections among boundaries present in the nine algorithms thus far presented, it is preferable to use the unifying approach to boundary generation which considers the two types of region roots involved and selects from the limited number of boundary types to find the new boundary. However, since there are infinitely many possible types of Snell's-Law edges based on the number of edge-crossings between the edge and the goal, an approximate solution is proposed. Since boundaries between Snell's-Law edges are similar to hyperbolas, it is proposed that for all except the varieties already derived in Appendix A, hyperbolas be used as approximations to the exact curves.

When a new boundary has been generated because of the intersection of two boundaries from different sets, the effects may propagate into both partial OPM's. This will be, in the worst case, a very expensive operation, because unlike Voronoi diagram construction, the boundaries between regions are not simple lines, and the effects are not guaranteed to be local. Each boundary which is truncated by the new boundary must be fol-

lowed to its end (before it was truncated), and if it intersected other boundaries, these in turn must be reconsidered with respect to the new boundary.

Algorithm VI-9 describes this method of constructing an optimal-path map for input maps containing any number of the seven types of primitive terrain features and connected river and road segments. At each level of recursion, the algorithm divides the terrain into two roughly equal sets, based on a calculation of the median leftmost vertex. At the lowest level, that of a single terrain feature, the algorithm calls on Algorithms VI-1 through VI-8 to construct an OPM for the feature. At higher levels, OPM's are merged by procedure **merge-opms**.

VII. ANALYSIS OF DIVIDE-AND-CONQUER EXACT-OPM ALGORITHM

A. SOURCES OF ERROR IN THE ALGORITHM

The divide-and-conquer exact-OPM algorithm produces a more accurate optimal-path map than the wavefront-propagation OPM algorithm, but it still has error with respect to the conceptual OPM it models. In terms of the categories of error discussed in Chapter II, the model-cost versus real-world-cost error occurs because of approximations in the terrain database of continuously-varying terrain with 12.5m square cells.

The second category of error, model-computed-cost versus model-optimal-cost, appears in several forms in the output of this algorithm. The two most significant are discussed here. First, each boundary whose analytical form does not have a closed-form solution is represented by a piecewise-linear approximation. These boundaries are plotted parametrically, iteratively setting one parameter and solving for the other. Fortunately for the precision of the algorithm, most boundaries have very little curvature (see for example, Figures 22, 23, and 24). An exact analysis of the impact of this type of error has not been done, but the proof-of-concept implementation for the high-cost, exterior-goal homogeneous-cost area (HCA) plotted twenty or fewer points for each curve, and in all test cases, error of this type was too small to be visible in the laser-printer output.

What error does occur will have the effect of causing start points which are close to a boundary to be placed in an incorrect region. These start points will then be associated with paths which are not quite optimal. But along a boundary there are two equal-cost paths to the goal from each start point. On an approximate boundary one of these two paths will be slightly more costly than the other. This error will be no greater than the cost-rate in the region times the maximum distance of the piecewise-linear approximation from the actual curve. Since the approximations seem to be very close to the actual curve in observed cases, it seems safe to state that this error can be ignored in most practical applications.

A second source of error in the category of model-computed-cost versus model-optimal-cost is using hyperbolas to approximate boundaries between homogeneous-behavior regions having paths with more than two Snell's-Law crossings. An exact analysis of the error caused here has not been done. But for regions whose paths have multiple Snell's-Law crossings leading to a region root which is a point, as the regions lie further and further away from the point, they have cost functions (called "distorted cones") which have flatter and

flatter iso-cost contours, leading to boundaries with less and less curvature. The approximating hyperbola should be closer and closer to the actual boundary as the boundary becomes almost linear. The error in the computed cost of an optimal path caused by this approximation can be ignored in most applications.

The third category of error discussed in Chapter II, that of model-computed-location versus model-optimal-location, occurs only in the situations discussed above where a start point is incorrectly placed on the wrong side of a boundary. When this happens, the computed path will have a distinctly different behavior than the true optimal path.

B. TIME AND SPACE COMPLEXITY OF THE ALGORITHM

We begin by analyzing the construction of the optimal-path tree, and then analyze the algorithms proposed for each primitive terrain feature type in an isolated setting, because the final algorithm uses the previous ones as base cases of its recursion.

1. Time and Space Complexity of Optimal-Path Tree Construction

Prior to the execution of the algorithms introduced in Chapter VI, the optimal-path tree (OPT) must be constructed. A brute-force method which finds optimal paths from each terrain-feature vertex and then inserts each path into the OPT would take, using the continuous-Dijkstra algorithm, $O(n^8L)$ time in the worst case, and using recursive wedge decomposition, $O(n^3)$ in the average case, where n is the number of terrain-feature vertices, and L is a measure of the precision of the problem representation. Insertion into the OPT as described in Chapter VI would take, in the worst case, no more than $O(n^2)$ time, because no path list is longer than n, and there are n path lists to be inserted. The optimal-path tree has no more than one node for each terrain-feature vertex and edge, plus one for the goal point. Thus, its worst-case space complexity is O(n), since with the assumed terrain constraints, there are O(n) edges. A more efficient way to use the continuous-Dijkstra algorithm is possible which computes paths to all vertices and builds the OPT in one execution of the algorithm, giving $O(n^7L)$ worst-case time complexity. Recursive wedge decomposition can also be modified to operate this way.

2. Time and Space Complexity of The Single-Obstacle-OPM Algorithm

Algorithm VI-1 constructs an optimal-path map for a single isolated convex obstacle with respect to a goal. For an obstacle with n vertices there are at most n shadow boundaries, which can be constructed in

O(n) time by a depth-first traversal of the the optimal-path tree, generating a shadow boundary for each node in the tree except the one representing the empty path list. Each hyperbola segment which is part of the opposite edge can be constructed in constant time, and there are at most n-2 intersections of the opposite-edge boundary with shadow boundaries. Thus the opposite-edge boundary can be constructed in O(n) time, so the entire OPM can be constructed in O(n) time. Each shadow boundary and each hyperbola segment of the opposite-edge boundary can be represented in O(constant) space. Since the optimal-path tree can be stored in O(n) space, and assuming constant accuracy, the representation of the entire OPM is O(n) space. (See Figure 17.)

3. Time and Space Complexity of The Single-River-Segment-OPM Algorithm

Algorithm VI-2 constructs boundaries generated by a single river segment. A river segment has exactly two shadow boundaries, at most two river-crossing boundaries, and exactly one river-obstacle boundary consisting of only one hyperbola segment. Thus there at most five boundaries to construct, each of which can be constructed in O(constant) time, so the time complexity of the algorithm is O(constant). Similarly, the space complexity is O(constant). (See Figure 18.)

4. Time and Space Complexity of The Single-Road-Segment-OPM Algorithm

Algorithm VI-3 constructs boundaries generated by a single road segment. By the analysis of Chapter V, a road segment may have at most fourteen boundaries, each of which can be constructed in O(constant) time, using O(constant) space. Thus the time and space complexity of Algorithm VI-3 are both O(constant). (See Figure 20 and 21.)

5. Time and Space Complexity of The High-Cost-Exterior-Goal-HCA-OPM Algorithm

Algorithm VI-4 constructs the planar partition for a region with higher cost than the surrounding terrain with a goal point outside the region. It has exterior boundaries which are similar in number to those generated by an obstacle, except that there may be as many as three opposite-edge boundaries. Thus, by the same reasoning as for obstacles, the construction of exterior boundaries has worst-case time and space complexity of O(n).

However, the interior boundaries are more time-consuming in the worst case, because of the way boundaries may intersect. (See Figures 22, 23, 24, 30, 31, and 32.) Each of the n HCA vertices is associated with an interior boundary. In the worst case, each pair of these boundaries intersects and a third boundary

begins at the intersection point, giving n/2 new boundaries, and each pair of new boundaries intersects and another begins, for n/4 new boundaries at the third level, and so on until a final boundary occurs which connects all the others. In this case, there are n+n/2+n/4+n/8+...+1 boundaries. There are in the limit n/(1-1/2) = 2n boundaries.

There does not appear to be a simple way to determine for a boundary which of the two adjacent boundaries will be paired with it. An iterative check which accomplishes this is outlined in procedure **pair-and-merge-bdrys** under Algorithm VI-4, (see Section A4 of Chapter VI and Figure 30). This procedure takes at worst (in a very pathological case), n-2 passes through the inner ("while PairedBdrySet is changing") loop, which itself processes n boundaries ("for all $B_{i,j}$..."). The outer ("while BdrySet is changing") loop, which checks for intersections by newly propagated boundaries with already-paired boundaries, could also take O(n) passes in the worst case. Thus procedure **pair-and-merge-bdrys** has worst-case time complexity of $O(n^3)$. This measure dominates the O(n) complexity of the exterior boundaries, and so the worst-case time complexity of Algorithm VI-4 is $O(n^3)$. The space complexity is O(n) because at most 2n interior boundaries, n-2 shadow boundaries, and n-2 portions of opposite-edge boundaries exist.

6. Time and Space Complexity of The High-Cost-Interior-Goal-HCA-OPM Algorithm

Algorithm VI-5 constructs the OPM for a high-cost HCA with an interior goal point. It has much lower time complexity than the high-cost, exterior-goal case, because the interior does not have a number of intersecting boundaries from which more boundaries may emanate. In fact, for each vertex, at most one exterior and four interior boundaries are generated, as well as additional boundaries for each pair of visible edges and each interior opposite-edge boundary. Both the exterior visible-edge boundaries and the exterior opposite-edge boundaries display the same behavior as obstacle opposite-edge boundaries, so that all of them together have no more than O(n) segments. The only iterative loop in the algorithm is the outer one which processes each of the n vertices, so the overall worst-case time complexity is O(n), as is the space complexity. (See Figure 25.)

7. Time and Space Complexity of The Low-Cost-Exterior-Goal-HCA-OPM Algorithm

Algorithm VI-6 constructs the OPM for a HCA of lower cost than the surrounding terrain and an exterior goal. This algorithm generates at most four boundaries per HCA vertex. Although there are interior boundaries similar to the high-cost, exterior-goal case where much computing effort was required to construct them,

in this case they are never mutually intersecting. Thus the entire algorithm has time complexity O(n). The space complexity is also O(n). (See Figure 27.)

8. Time and Space Complexity of The Low-Cost-Interior-Goal-HCA-OPM Algorithm

Algorithm VI-7 constructs the OPM for a HCA with lower cost than the surrounding terrain, and a goal inside the HCA. This is the simplest of the four HCA cases, because there are exactly two linear boundaries emanating from each HCA vertex. Thus the time and space complexity is O(n). (See Figure 26.)

9. Time and Space Complexity of The Multiple-Connected-River-Segment-OPM Algorithm

Algorithm VI-8 constructs an OPM for multiple connected river segments. The time complexity of this algorithm depends on how many "event points" and new boundaries occur at each segment. An event point occurs on a river segment at each place that a boundary intersects it and denotes a point at which the algorithm must check for a continuation of the boundary on the other side of the segment. Since a river segment's boundaries will only intersect river segments in its shadow, the worst-case time complexity happens when the river "doubles back" on itself. Consider a sequence of connected river segments as in Figure 34. In this example, the closest two river segments to the goal, and each subsequent pair of segments, are positioned so as to cast two shadow boundaries which create event points on the next segment. Since in this example, the cost of the river is so small that each river-crossing boundary begins "outside" any event points on the segment and does not intersect any shadow boundaries, the shadow boundaries all continue to the next level of river segments. At the first level, four boundaries begin, and at each subsequent level, there are three new boundaries plus the continuation of boundaries from previous levels associated with event points. The result is that on each river segment, say at level i, there are 3i+1 possible boundaries generated. So for a sequence of river segments with n vertices, it is possible to have $\sum_{i=1 \text{ to } n/2} (3i + 1) = 3n^2/8 - 7n/4$ total boundaries over the entire set. Thus the worst-case time complexity of Algorithm VI-8 is $O(n^2)$. Since there are $O(n^2)$ boundary segments, the worst-case space complexity is also $O(n^2)$.

10. Time and Space Complexity of The Multiple-Feature-Divide-and-Conquer-OPM Algorithm

Algorithm VI-9 is the algorithm which takes OPM's for individual decomposable terrain features and merges them into one OPM. It uses the divide-and-conquer paradigm, and spends O(n) time dividing the map at each stage of size n, by standard median-finding algorithms from computational geometry. Let the time complexity of the algorithm itself be expressed as T(n). Then the recursive application of the algorithm to both

halves of the map will take 2T(n/2) time. Thus the dividing, recursion, and merging will take $T(n) = O(n) + 2T(n/2) + O(f(n))$, where f(n) is the time complexity of the merge step.

The procedure **merge-opms** is very similar to procedure **pair-and-merge-bdrys** associated with Algorithm V-4 for high-cost, exterior-goal HCA OPM's, which joined the interior boundaries and propagated new ones as needed. It is subject to the same possibility that multiple levels of newly-propagated boundaries may occur, and has the added complexity that for each boundary truncated in one of the subordinate OPM's, the procedure **intersect-and-merge** must be performed to reconstruct any other boundaries which previously intersected the truncated boundary but no longer do so. By the same reasoning as paragraph 4 above, even assuming that procedure **intersect-and-merge** has O(constant) worst-case time complexity, procedure **merge-opms** operates in $O(n^3)$ time. In fact, procedure **intersect-and-merge** operates in O(n) time in the worst case, because there are at most O(n) boundaries which a boundary can possibly intersect. Thus, procedure **merge-opms** has worst-case time complexity $O(n^4)$. We note also that the base case of the recursion requires the solution of a single-terrain-feature algorithm, which may have as much as $O(n^3)$ time complexity. Thus the worst-case time complexity of the entire algorithm may be stated as $T(n) \leq O(n) + 2T(n/2) + O(n^4)$, or

$$T(n) \leq 2T(n/2) + O(n^4)$$

for $T(1) \leq O(m^3)$, where m is the largest number of terrain-feature vertices which occur in a high-cost, exterior-goal HCA. Expanding this recurrence relation, gives, by induction on the depth i of the recursion, that for some constant c_1,

$$T(n) \leq 2^i T(n/2^i) + c_1 n^4 (1 - 1/2^{3(i-1)}).$$

Let $n = 2^k$, assuming that k is an integer. Then at the last splitting step, i=k, and we have that

$$T(n) \leq 2^k T(1) + c_1 n^4 (1 - 1/2^{3(k-1)}).$$

But for the base case, we have that $T(1) \leq c_2 m^3$ for some constant c_2, so

$$T(n) \leq 2^k c_2 m^3 + c_1 n^4 (1 - 1/2^{3(k-1)})$$

$$T(n) \leq c_2 n m^3 + c_1 n^4 - 8\, c_1 n.$$

Since $m \leq n$, Algorithm VI-9 has worst-case time complexity $T(n) = O(n^4)$.

C. EMPIRICAL PERFORMANCE OF THE ALGORITHM IMPLEMENTATION

The high-cost, exterior-goal case, was implemented as a proof-of-concept program. The high-cost, exterior-goal HCA was chosen because it was the most complex of the seven cases and incorporated most of the types of boundaries. The implementation was not intended to be particularly efficient, but was primarily designed to corroborate the shapes of various boundaries when compared with multiple runs of a point-to-point weighted-region path-planning implementation by Richbourg [Ref. 21]. Figures 22, 23, and 24 represent results of the OPM implementation overlaid on vectors representing the initial directions of a dense sampling of optimal paths from Richbourg's "Snell's Law" program. OPMs of fairly simple complexity such as the above three figures took four to six minutes apiece to construct, not counting the time necessary to find optimal paths from each terrain-feature vertex using Richbourg's point-to-point path-planner [Ref. 21]. This implementation was done in C-Prolog on a VAX 11/785 running under BSD 4.3 Unix.

VIII. CONCLUSIONS

A. GENERAL

In this research we developed two approaches to the construction of a planar partition for optimal-path maps (OPM). The first is an extension of the grid-based *wavefront propagation* algorithm for point-to-point path planning, for which we implemented and analyzed three versions. The second is based on *spatial reasoning* about how optimal paths behave in the presence of terrain features, leading to a divide-and-conquer algorithm. We assume that paths lie in free terrain consisting of five types of regions: homogeneous-cost background, convex polygonal obstacles, piecewise-linear rivers with a fixed crossing cost, piecewise-linear roads with a constant cost-rate, and convex homogeneous-cost areas. Additionally, we assume that no two features share a vertex. We assume that the mobile agent is of negligible size with respect to the surrounding terrain, and that the terrain is fixed and known.

Point-to-point path-planning algorithms require anywhere from $O(n^2 \log n)$ time for binary terrain (visibility-graph methods [Ref. 1]) to $O(n^7 L)$ time for homogeneous-cost areas (continuous-Dijkstra algorithm [Ref. 15]), where n is the number of terrain-feature vertices and L is a measure of the precision of the problem representation. One way to decrease the amount of run-time complexity of path-planning at the expense of increased preprocessing time and increased storage requirements is to construct optimal-path maps (OPM) which group optimal paths from *all* start points on a map with respect to a goal point by partitioning the plane into regions whose paths behave similarly. At run-time standard point-location techniques from computational geometry can be used to locate a start point in a region of the OPM in $O(\log n)$ time, and the optimal path can be reconstructed based on the known behavior of paths in the region.

B. COMPARISON OF WAVEFRONT-PROPAGATION TO SPATIAL- REASONING APPROACHES TO OPM CONSTRUCTION

The spatial reasoning approach to optimal-path-map construction is clearly preferable to wavefront propagation for applications requiring low error in the cost of the solution path compared with the cost of the actual optimal path. Otherwise, the wavefront-propagation approach using the diverging-path version seems

preferable because it does not depend on the labeling of vertex or edge cells, and is simpler than the exact algorithm, when the cost of constructing the optimal-path tree is included. The most accurate wavefront-propagation OPM algorithm, the vertex-edge version, requires an additional preprocessing phase which fits polygons to grid-based terrain features and assigns vertex and edge labels to cells. This terrain preprocessing is also necessary in the spatial-reasoning approach used on large-scale cross-country terrain data, because Defense Mapping Agency provides data in the form of 25 meter or 12.5 meter square grid cells from which the polygonal terrain features of the spatial-reasoning approach must be derived. Since implementation of wavefront propagation is simpler than the exact-OPM divide-and-conquer algorithm, it may be preferable in applications which can afford the 7.6% inaccuracy to use the vertex-edge version of wavefront propagation.

While wavefront propagation would seem to be preferable if accuracy is not a factor, it should be noted that the complexity of wavefront propagation is based on the number of cells in the input map, not the number of terrain-feature vertices, so the two time complexity measures are not precisely comparable. However, for a grid-based map of O(m) cells, with a corresponding polygonal map of v vertices, if it could be said that the frequency with which a cell includes a vertex would be constant as the size of the map increased, v would increase linearly as a function of m. By this reasoning, we could expect a typical polygonal map for a grid with m cells to have O(m) vertices, so the measures are approximately comparable.

Actual average performance could give different results from worst-case analysis. Since the spatial-reasoning-OPM divide-and-conquer algorithm was implemented only for one of the seven cases as a test-of-concept instrument, actual performance tests of the exact-OPM algorithm were not possible.

C. USEFULNESS OF THE OPM APPROACH TO PATH PLANNING

Since the OPM approach to path planning trades preprocessing time and increased storage for improved speed at run-time, it will be useful in applications which require real-time response to a path-planning query, such as autonomous-vehicle or missile path-planning, or where multiple queries over the same terrain are expected, for example, in a terrain-analysis decision aid for tactical military units.

Two major objections to the OPM approach are its preprocessing time and its storage requirements. Certainly preprocessing will take longer than current path-planning methods. However, the non-automated approach to terrain navigation in many domains, which has been to prepare paper maps well ahead of time for

distribution to users, could serve as a model for OPM preprocessing, wherein an organization such as Defense Mapping Agency could devote centralized resources to the preprocessing phase and distribute standard OPM databases so that field units or vehicles would have to devote resources only to the run-time phase.

A second objection to the OPM approach is the need for increased storage. However, the cost and compactness of storage media is constantly being reduced by research and development efforts. OPM databases could be recorded on optical disks or "digital paper", allowing space for a whole array of OPMs covering an approximation of the four-dimensional solution for a given geographical area. A typical OPM for an area of 20 by 20 kilometers might include on the order of 800,000 boundary segments (100 vertices per square kilometer times 400 square kilometers giving on the order of 40,000 boundaries, times 20 segments per boundary), each requiring two points of two coordinates each, or 3.2 megabytes of storage. For a four-dimensional array of OPM's representing all optimal paths from any start point to a sampling of perhaps 10 goal points per square kilometer, or 4,000 OPMs, 12.8 gigabytes would be required. As of 1989, 5-1/4-inch disks using digital-paper technology are commercially available which store 1 gigabyte each [Ref. 45]. The approximately thirteen such disks needed to store a full set of OPMs would be easily transportable. A library of OPMs for various potential areas of operation could be maintained, for example, much as libraries of paper maps are maintained.

D. AREAS FOR ADDITIONAL RESEARCH

The terrain types assumed herein do not include non-convex polygons, even though much real-world terrain would be difficult to model accurately without them. Thus, it is important to determine how to incorporate non-convex polygons into the optimal-path map algorithms presented. With the unifying view of regions and boundaries based on region cost functions, this task seems attainable with additional research.

The boundary between regions where one or both regions have paths which cross multiple Snell's-Law edges en route to a region root which is a point has not been characterized analytically. In the current algorithm, it is proposed that such boundaries be approximated by hyperbolas, and it is thought (without proof) that such an approximation introduces very little error. However, a better approximation could be used to intersect with other cost functions to determine boundaries on a much less ad hoc basis than is done in this dissertation.

One specific place in which improvement in efficiency could have great effect on the overall exact-OPM algorithm is in constructing the interior boundaries of an exterior-goal, high-cost HCA in less than $O(n^4)$ time. OPM's for all six other primitive terrain-features can be constructed in O(n) or less time, and for multiple connected river segments in $O(n^2)$ time, and it is this single case which drives the divide-and-conquer algorithm's worst-case time complexity to $O(n^4)$. In addition, a merge procedure for the exact-OPM divide-and-conquer algorithm which had efficiency more in line with that of Voronoi diagram construction would improve overall performance.

A four-dimensional solution is needed in order to make the OPM approach useful in most domains. The solution consistent with the approach herein is to create multiple OPM's for a sampling of goal points in the plane, and then choose the OPM to use at run-time based on the proximity of the query goal point to the goal point of one of the OPM's. Perhaps more efficient methods exist which would characterize boundaries between four-dimensional regions in a space of all start and goal points, a conceptual generalization of the two-dimensional solutions reported here for start points and a fixed goal. In other words, the four-dimensional hyperplane would be partitioned into regions whose paths were similar.

It would be very instructive, as well as practical, to implement a complete two-dimensional path-planning system, from construction of an optimal-path tree for the four types of terrain used herein through OPM construction, and including a run-time system to accomplish point location and path reconstruction.

APPENDIX A - THEOREMS

A. OVERVIEW

In this appendix the theorems which form the basis of the research reported herein are presented, along with associated lemmas, corollaries, and fundamental assumptions. The theorems follow in the same order in which they are discussed in the body of this report, and are numbered by chapter and theorem. Lemmas and corollaries are numbered as extensions of the theorem to which they apply. First, some notation used in this appendix and throughout the report is presented. Then three theorems and a fundamental assumption with three associated corollaries are presented which provide a theoretical foundation for the discussions of Chapter I. Next six theorems are presented which state the basic boundary equations as developed by the unifying view of region cost functions. Seven theorems from Chapter V, one for each of the three terrain-feature types obstacle, road segment, and river segment, and four for the four cases of the homogeneous-cost area, are presented. The definition of homogeneous-behavior region used in this appendix is the set of all points whose optimal paths have the same path list.

B. NOTATION

The following notation is introduced for use with respect to path-planning.

Example	Description
P	A point in Euclidean n-space.
$\overline{PQ}$	The straight-line segment from P to Q
(PQ)	A feasible path from P to Q
$(PQ)_i$	The ith feasible path from P to Q
(PQ)*	Optimal path from P to Q
OPL(P)	Optimal-path list (sequence of edges and vertices encountered) of P.
OPL(P) = [P,Q|OPL(Q)]	The path list from P through Q shown in Prolog-style list notation (i.e., lists are enclosed in braces, commas separate elements, and the entry following a vertical line ("|") is the "rest" of the list).
$\|(PQ)_i\|$	The cost (weighted distance) from P to Q via path $(PQ)_i$.
d(P,Q)	The Euclidean distance between P and Q.
$((PQ)_i(QR)_j)$	A feasible path from P through Q to R.

$(PQ)_i \subset ((PQ)_i(QR)_j)$	Set notation applies to paths as if to their path lists, treating them as ordered sets, e.g., $(PQ)_i$ is a sub-path of $((PQ)_i(QR)_j)$, but $(QP)_i$ is not a sub-path of $((PQ)_i(QR)_j)$.
$P \in (PQ)_i$	Points are considered elements of paths.
$(PQ)_i = (RS)_j$ iff $(PQ)_i \subseteq (RS)_j$ and $(RS)_j \subseteq (PQ)_i$	Two paths are equal if $\forall k$, the kth elements of the path lists of the two paths are the same.
c_{pq}	The cost (weighted distance) of a path from point P to point Q
r_i	The cost rate in region i.
θ_1	Angle of incidence or refraction of a path across a Snell's-Law edge.
$\psi = \sin^{-1}(r_1/r_2)$	Critical angle with respect to a Snell's-Law edge separating regions of cost-rates r_1 and r_2, where $r_1 < r_2$.
∇AGB	The characteristic wedge with vertex at G and edges through A and B. This is defined with respect to road segments such that G is the goal point, A and B are points on the line of the road segment, ray GA forms angle $\pi/2 + \psi$ with the segment, and ray GB forms an angle $\pi/2 - \psi$ with the segment, where ψ is the critical angle as defined above.

C. BASIC THEOREMS

THEOREM I-1. Given optimal path $(AB)^*$, $\forall P \in (AB)^*$, $(PB)_i = (PB)^*$ if $(PB)_i \subset (AB)^*$, i.e., any sub-path of an optimal path is also an optimal path. (The generalization of this concept is known in some contexts as the *principle of optimality*, the *dynamic programming principle*, or the *Markovian property* [Ref. 46].)

PROOF I-1: (Proof by Contradiction) Given points A and B and path $(AB)_1 = (AB)^*$ such that $|(AB)^*| = c^*$, points P and Q such that $P \in (AB)^*$ and $Q \in (AB)^*$, where paths (AP), (PQ) and (QB) are such that $((AP)(PQ)(QB)) = (AB)^*$ with $|(PQ)(QB)| = c_1$, and $Q' \notin (AB)^*$. (See Figure 35.)

Assume $\exists$ (PQ') and $(Q'B)$ such that $|((PQ')(Q'B))| = c_1'$, and $c_1' < c_1$. Then $\exists$ $(AB)_2 = ((AP)(PQ')(Q'B))$ such that $|(AB)_2| = c^* - c_1 + c_1'$. But $c^* - c_1 + c_1' < c^*$, so $|(AB)_2| < |(AB)^*|$, which contradicts the optimality of $(AB)^*$. ♦

THEOREM I-2. In terrain consisting of a homogeneous-cost background on which is placed homogeneous-cost polygons, optimal paths change directions only at terrain feature vertices and edges. Note that the terrain defined in Chapter II, Section E, are specializations of this type of terrain. (See Figure 36.)

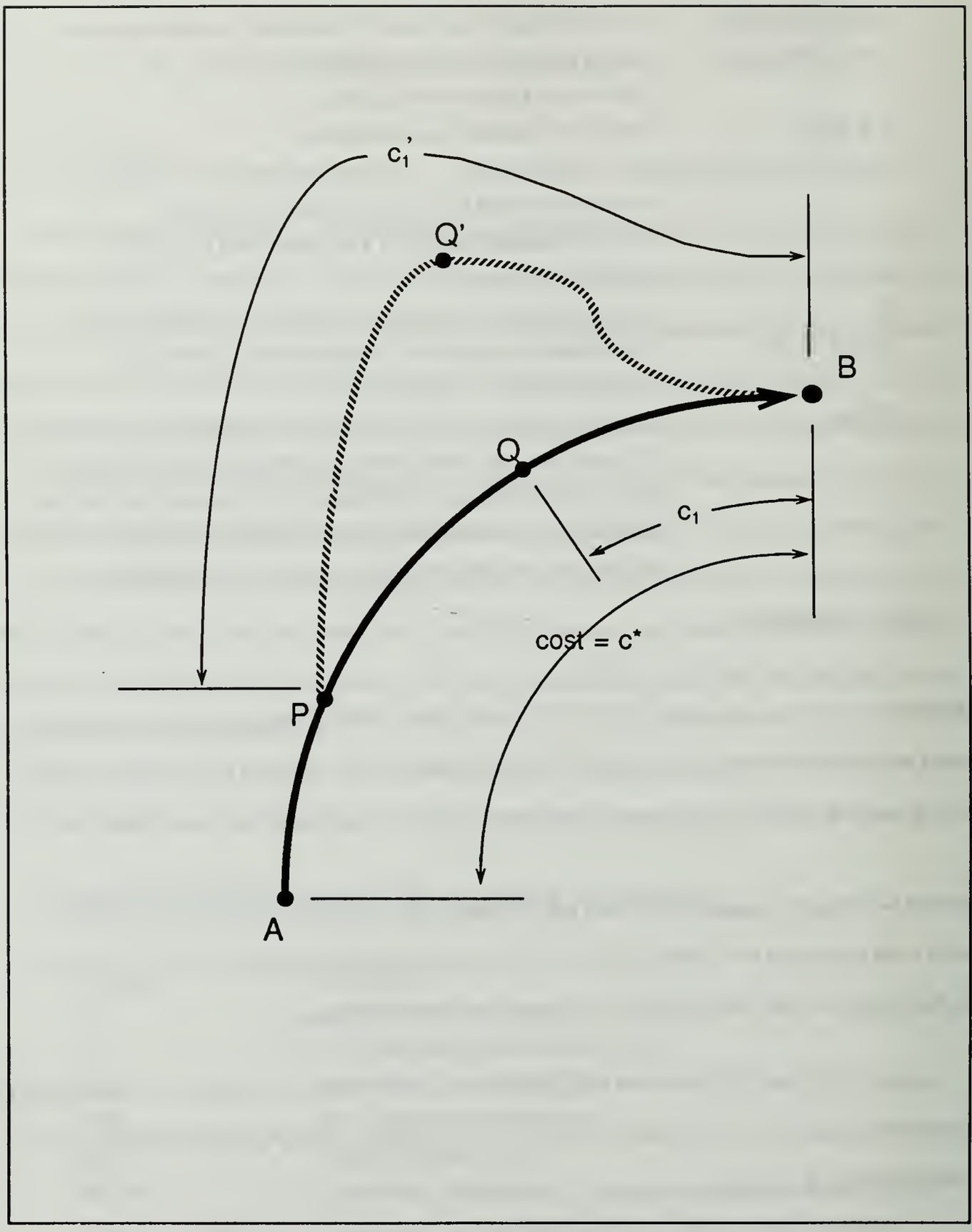


Figure 35
Principle of Optimality for Path Planning

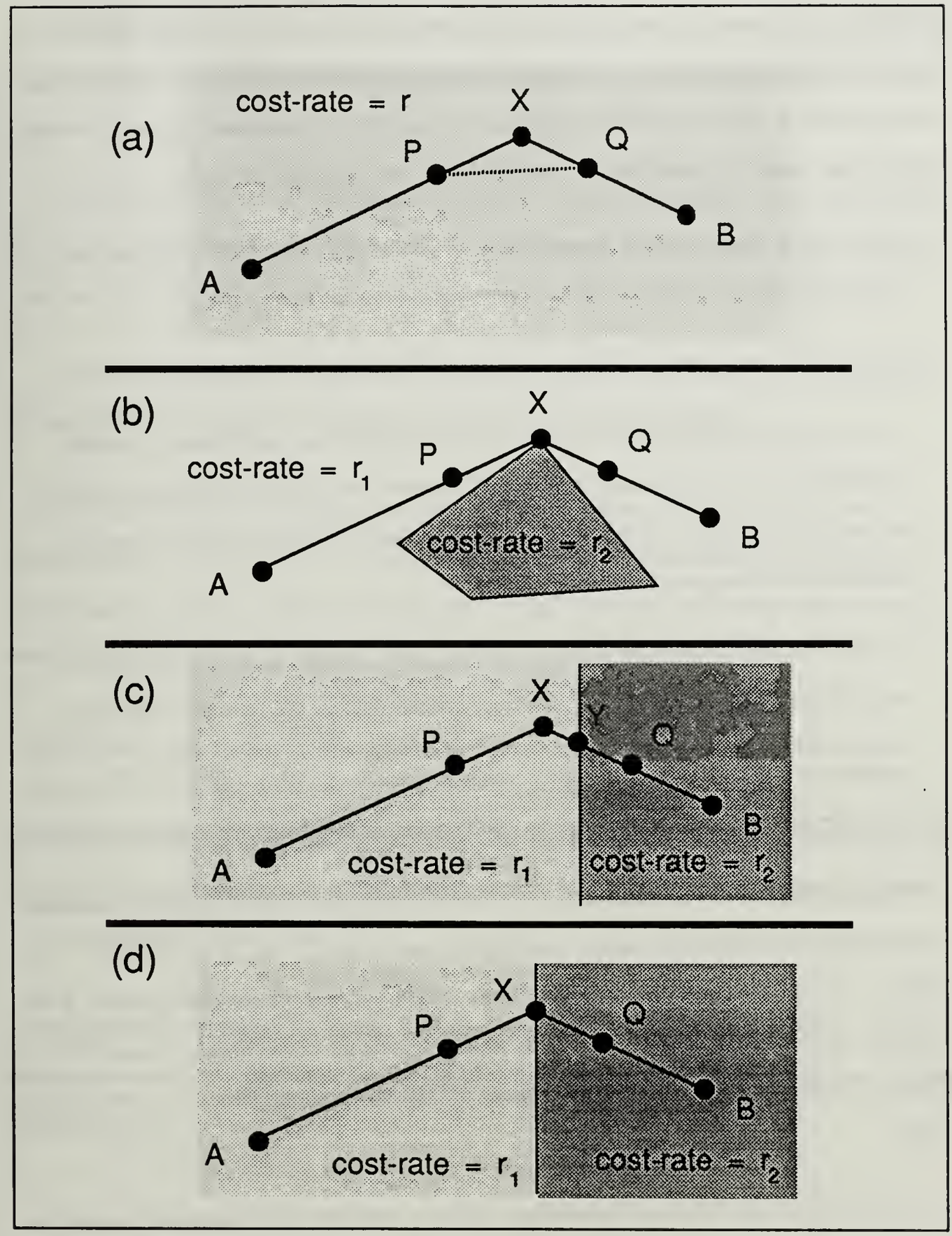


Figure 36
Optimal-Path Turn Points

PROOF I-2: Consider point X on optimal path (AB)*, with P$\in$ (AB)* and Q $\in$ (AB)* arbitrarily close to X such that ((PX)(XQ)) $\subset$ (AB)*, i.e., P and Q are on opposite sides of X on path (AB)*. Assume P, X, and Q are not colinear (i.e., X is a turn-point). Among terrain consisting of line segments and polygons, P and Q can be made close enough to X so that there are only four possible placements of P, X, and Q:

(1) P, X, and Q are in areas of equal cost, X is not coincident with a terrain feature vertex, and line segment $\overline{PQ}$ does not intersect any terrain feature edge.

(2) P, X, and Q are in areas of equal cost, and X is coincident with a terrain feature vertex.

(3) X is in an area of equal cost with either P or Q, but not both. Assume without loss of generality that P and X are in an area of cost r_1 and Q is in an adjacent area of cost r_2. Additionally, X is not on a terrain feature edge, (PX) does not cross any edges, and (XQ) crosses exactly one edge, the edge between the two areas of concern.

(4) X is on the terrain feature edge separating an area of cost r_1 of which P is a member and adjacent area of cost r_2 of which Q is a member. Additionally, neither (PX) nor (XQ) cross any other edges.

Assume case 1. ((PX)(XQ)) = (PQ)* by the principle of optimality. So $|(PX)| + |(XQ)| \leq |\overline{PQ}|$, because of the optimality of ((PX)(XQ)) (i.e., the cost from P to Q via X is less than the straight-line cost from P to Q).

So it is also true that $|(PX)|/r + |(XQ)|/r \leq |\overline{PQ}|/r$. Now $\forall$R and S, the Euclidean distance between R and S is less than or equal to the distance along any general path between R and S. So $|(PX)|/r \geq |\overline{PX}|/r$ and $|(XQ)|/r \geq |\overline{XQ}|/r$. (By the notational convention that |(RS)| is the *weighted* distance, or cost, between R and S, |(RS)|/r is the Euclidean distance of (RS) if (RS) lies entirely in an area of cost rate r.) Therefore $|\overline{PX}|/r + |\overline{XQ}|/r \leq |\overline{PQ}|/r$. But since P, X, and Q are not collinear, this violates the triangle inequality, so case 1 is not possible.

It is clear, by example, that case 2 is possible. Consider X coincident with the corner of a rectancular obstacle O, with P and Q not intervisible, but closer to X than to any other vertex of O. $(\overline{PX}\ \overline{XQ})$ = (PQ)* in this case, demonstrating that case 2 is possible, i.e., that optimal paths may turn at terrain-feature vertices.

Assume case 3. Let Y be the point at which (XQ) crosses the edge. Then by the same reasoning as for case 1 above, it is contradicted that P, X, and Y are not collinear, i.e., X is not a turn-point, so case 3 is not possible.

Richbourg [Ref. 20] proves the applicability of Snell's Law to describe the angles of incidence and refraction of an optimal path across an edge as in case 4, demonstrating that this case is possible, i.e., that optimal paths may turn as they cross terrain-feature edges.

Thus the only turn-points in optimal paths in terrain consisting of homogeneous-cost polygons on a homogeneous-cost background are coincident with terrain feature vertices or edges. ♦

ASSUMPTION I-3, General-Position Assumption: No terrain-feature vertex or edge interior lies on a non-trivial homogeneous-behavior-region boundary, i.e., a homogeneous-behavior boundary other than those of the homogeneous-behavior region of which the vertex or edge is the root, or the terrain-feature edges incident upon the vertex or edge.

COROLLARY I-3.1: There is a unique optimal path from each terrain-feature vertex and edge interior.

PROOF I-3.1: (Proof by Contradiction) Assume that there were two optimal paths from a terrain-feature vertex or edge interior. Then the vertex or edge would lie on a non-trivial boundary, by the definition of a boundary. But this contradicts Assumption I-3. ♦

COROLLARY I-3.2: There is a unique homogeneous-behavior region root associated with each homogeneous-behavior region, where a region root is the first vertex or edge crossed by optimal paths which start in the region.

PROOF I-3.2: From the definition of a homogeneous-behavior region as the set of points whose optimal paths to a goal point have the same path list, the path lists from all start-points in a region are identical, so the first elements of the path lists are also identical. Thus there is only one root per homogeneous-behavior region. Assume there existed two homogeneous-behavior regions which shared the same root. Since a region consists of all points with identical optimal-path lists, then $\exists OPL_1 = [E_1|Rest_1]$ and $\exists OPL_2 = [E_1|Rest_2]$ such that $Rest_1 \neq Rest_2$. By the definition of a boundary, E_1 would thus be on the boundary between region 1 and region 2. By Theorem I-2, E_1 must be a terrain-feature vertex or edge, but this contradicts the general-position assumption. Thus there is only one homogeneous-behavior region per root. ♦

DEFINITION I-3.3: A region R is star-shaped if $\exists P \in R$ such that $\forall Q \in R$ and $\forall X \in \overline{PQ}$, $X \in R$.

COROLLARY I-3.4: Homogeneous-behavior regions are star-shaped with respect to their region roots.

PROOF I-3.4: By the definition of a homogeneous-behavior region, all start-points in the region have the same optimal-path list, with, by the definition of a region root, the same first element. By Theorem I-2, the optimal path from each start-point to the root is a straight line segment. By the Theorem I-1, all points along the line segment have optimal paths lying along the line segment, so sharing the first element of their optimal-path lists as well, and so by Corollary I-3.1 sharing optimal-path lists. Thus all points along each such line segment lie in the same homogeneous-behavior region. ♦

THEOREM I-4: Given a two-dimensional map of a finite number of linear and polygonal terrain features and a goal-point, it has a unique optimal-path tree.

PROOF I-4: Given a two-dimensional map M of linear and polygonal terrain features and a goal G, each point S in M either has an optimal path, i.e., the feasible path of minimum cost, or else has no feasible path to G. If it has an optimal path, then by the definition of an optimal-path list and Theorem I-2, it also has an optimal-path list. If it has no feasible path, it is associated by convention with the optimal-path list [[],G], where [] represents the null list. Define the relation $R = \{(P_1,P_2) \mid OPL(P_1) = OPL(P_2)\}$, i.e., two points are related by R

if and only if their optimal-path lists are identical. Since identity is an equivalence relation, so is R, so R completely partitions the plane into sets of points with identical optimal-path lists. Since this is the definition of a homogeneous-behavior region, the plane is completely partitioned into homogeneous-behavior regions. Since there are a finite number of terrain-feature vertices and edges, there are a finite number of homogeneous-behavior regions.

A directed acyclic graph can be used to represent a partial order among its nodes [Ref. 36]. A partial order of a set S is a binary relation U such that $\forall$ a $\in$ S, aUa is false , i.e., U is irreflexive, and $\forall$a, b, and c $\in$ S , if aUb and bUc, then aUc , i.e., U is transitive. [Ref. 36] The set of all homogeneous-behavior regions in map M is partially ordered by their optimal-path lists as follows. Let U = $\{(P_1,P_2) \mid P_1 \subset P_2\}$, i.e, optimal-path list P_1 precedes optimal-path list P_2 in the partial order if P_1 is a proper subset of P_2. Because the relation "proper subset" induces a partial order on a set whose elements are sets, the relation U also induces a partial order on the set of optimal-path lists, and hence on the set of homogeneous-behavior regions, of map M with respect to goal G. In fact, because of the uniqueness of optimal-path lists from region roots, a specialization of the directed acyclic graph, the tree, may be used to represent the partial order of homogeneous-behavior regions. We call this tree an optimal-path tree, because it represents the optimal paths of map M.

Now consider the homogeneous-behavior regions in M with optimal-path lists consisting of only one element. Since all optimal-path lists for optimal paths to G have by definition the point G as their last point, and by the definition of homogeneous-behavior regions as the set of points with identical optimal-path lists, there is only one region with a single element in its optimal-path list, the region with the optimal-path list [G], and [G] is a subset of all other optimal-path lists. Thus [G] precedes all other optimal-path lists in the partial order, and so is the root of optimal-path tree $T_{M,G}$, the optimal-path tree associated with map M with respect to goal point G. ♦

D. PROOFS FOR BASIC BOUNDARY EQUATIONS

THEOREM V-0.1: *(Boundary between two regions with paths which go initially to two different points)* Given goal point G and two adjacent homogeneous-behavior regions of cost rate r whose region roots are points V_1 and V_2, costs $c_1 = |(V_1G)^*|$ and $c_2 = |(V_2G)^*|$ (the costs of optimal paths from V_1 and V_2 respectively) where without loss of generality it is assumed that $c_2 > c_1$, the boundary between regions 1 and 2 is a portion of the hyperbola branch which is closer to V_2 than to V_1, and is described by

(Equation 1) $$\frac{x^2}{a^2} - \frac{y^2}{b^2} = c^2$$

where $a = (c_2 - c_1)/2$, $c = r\, d(V_1,V_2)$, and $b^2 = c^2 - a^2$, and where the x-axis is oriented along the line segment $\overline{V_1V_2}$ with the origin at a point half-way between V_1 and V_2 .

PROOF V-0.1: (See Figure 37.) By the definition of a homogeneous-behavior region, points in region 1 all have the same path list, whose first element is V_1. Thus the first leg of an optimal path from any point P in region 1 is $\overline{PV_1}$. Similarly, the first leg from any point P in region 2 is $\overline{PV_2}$. The boundary between regions 1 and 2 is the set of points P such that $c_1 + |\overline{PV_1}| = c_2 + |\overline{PV_2}|$. Therefore $|\overline{PV_1}| - |\overline{PV_2}| = c_2 - c_1$. From basic analytical geometry, the set of points with constant absolute difference of distances from two foci is a hyperbola. Since the above equation describes the signed difference of the two distances, it represents one branch of the hyperbola, the branch such that $|\overline{PV_1}| > |\overline{PV_2}|$. Thus the branch on which P lies is closer to V_2 (the vertex with the higher-cost optimal path) than to V_1. ♦

THEOREM V-0.2: *(Boundary between a region with paths which go initially to a point, and a region with paths which go to and travel along a linearly-traversed-edge, or "road")* Given goal point G and two adjacent homogeneous-behavior regions with cost-rate r_0, one region having point U as root and the other having linearly-traversed edge $\overline{VW}$ as root, where $\overline{VW}$ is a sub-segment of some terrain-feature edge such that OPL(V) = [W | OPL(W)] and the cost-rate along the edge is r_{vw}, (for example, a road segment where paths leave the road from point W), the boundary between them is a portion of the curve

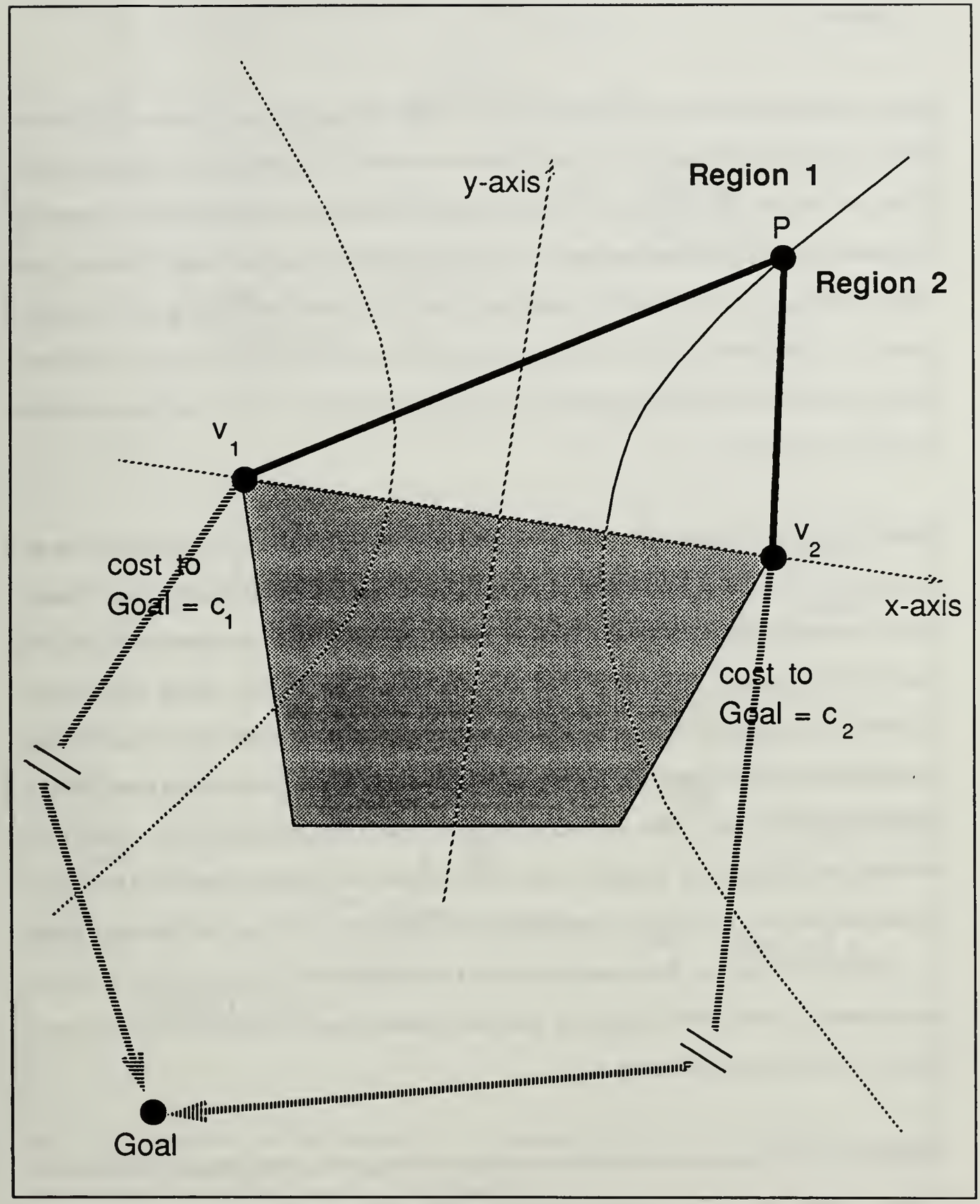


Figure 37

Boundary Between Homogeneous-Behavior Regions with Point Roots

(Equation 2) $y^2 = 4\,p\,x$,

where p is defined as follows. From W extend a ray $\overline{WW_d}$ away from region 2 (i.e., no point on $\overline{WW_d}$ lies in region 2) such that $\angle VWW_d = \pi/2 + \psi$, and the distance between W and W_d is $(c_w - c_u)/r_0$. Let point U_d be the point such that line $\overline{UU_d}$ is parallel to $\overline{WW_d}$, and the line $\overline{U_d W_d}$ is perpendicular to line $\overline{UU_d}$. Let point O be the point on line $\overline{UU_d}$ equidistant between U and U_d. Then the coordinate axes are the line $\overline{UU_d}$ (x-axis with U in the positive x direction) and the line through O parallel to $\overline{U_d W_d}$ (y-axis with W_d in negative y direction), and $p = (c_w - c_u)/r_0$, where $\psi = \sin^{-1}(r_{vw}/r_0)$ is the *critical angle*, $c_w = |(WG)^*|$, and $c_u = |(UG)^*|$ (the costs of optimal paths to goal point G from W and U respectively). Note that the x-axis is the parabola axis and the line $\overline{U_d W_d}$ is the directrix.

PROOF V-0.2: (See Figure 38.) The boundary between regions 1 and 2 is the set of points P such that the cost of optimal paths which go through U and through W are the same. The optimal path through U begins with the line segment $\overline{PU}$ and continues with $(UG)^*$ and has total cost c_u, while the optimal path through W starts with the line segment $\overline{PQ}$ at cost-rate r_0, where Q is a point on $\overline{VW}$ between V and W inclusive, continues along line segment $\overline{QW}$ at cost-rate r_{vw}, and ends with path $(WG)^*$ with total cost c_w. Thus, the boundary is described by the equation $r_0\, d(P,U) + c_u = r_0\, d(P,Q) + r_{vw} d(Q,W) + c_w$, or rearranging terms, $d(P,U) = d(P,Q) + \sin\psi\, d(Q,W) + (c_w - c_u)/r_0$. Now $\angle PQW = \pi/2 + \psi$ for a road, as shown by Rowe [Ref. 2]. Extending the line $\overline{PQ}$ to point P_d, as Figure 38 shows, the right-hand side of the above equation is the straight-line distance from P to P_d . Let line D lie perpendicular to $\overline{PQ}$, through P_d. By Figure 38, line D is a distance $(c_w - c_u)/r_0$ from W. Thus, the above equation states that P is equidistant from point U and line D, the definition of a parabola with the form of Equation 2, where the coordinate axes are the lines $\overline{UU_d}$ and D as shown, and p is half the distance from U to U_d. ♦

THEOREM V-0.3: *(Boundary between regions having paths which go to and travel along two different linearly-traversed edges, or "roads")* Given goal point G and two adjacent homogeneous-behavior regions with cost-rate r_0, one region having linearly-traversed edge $\overline{XY}$ as root and the other having linearly-traversed edge $\overline{VW}$ as root, where $\overline{XY}$ and $\overline{VW}$ are sub-segments of terrain-feature edges such that OPL(X) = [Y | OPL(Y)],

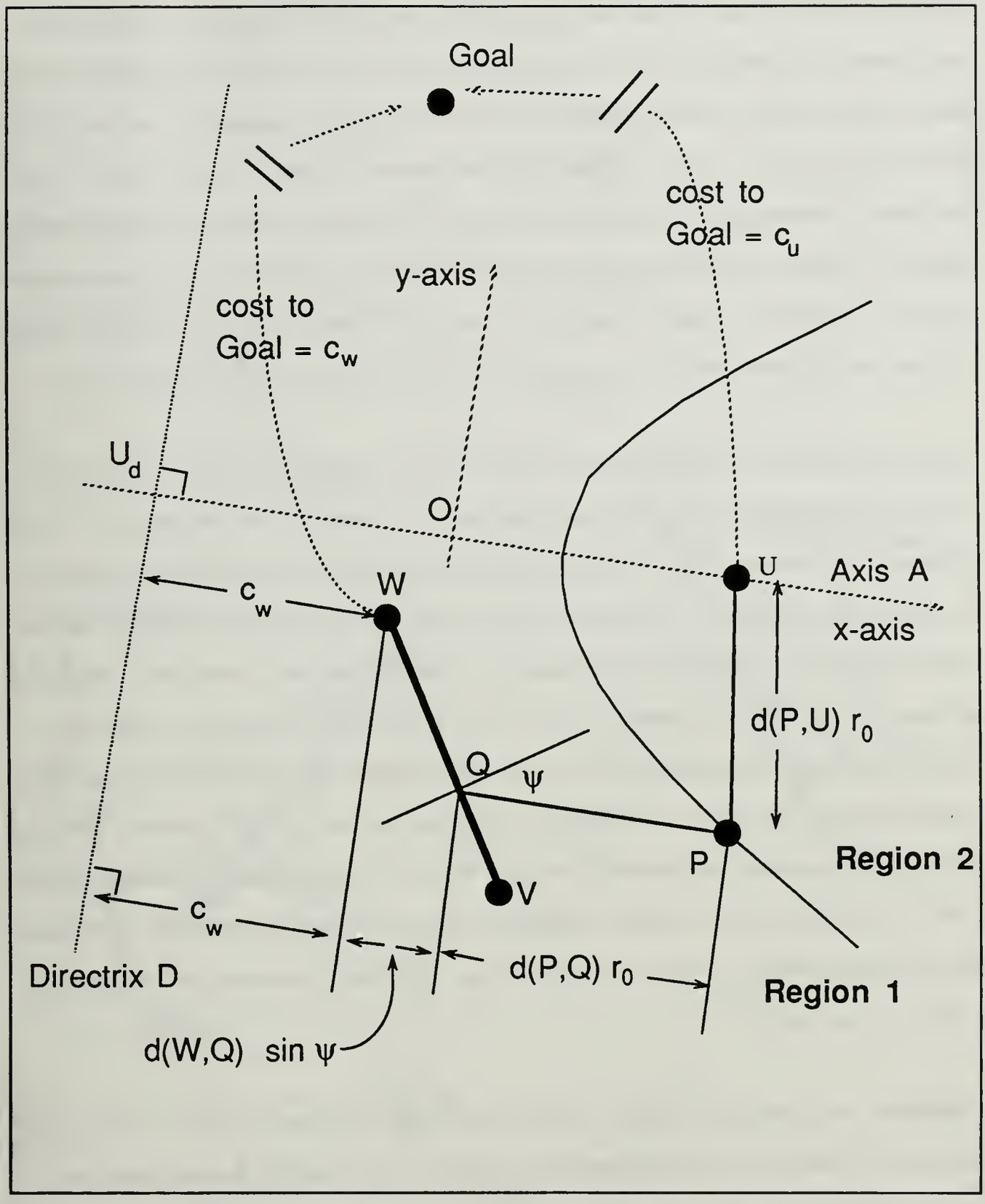


Figure 38

Boundary Between Homogeneous-Behavior Regions with a Linearly-Traversed Root and a Point Root

OPL(V) = [W | OPL(W)] and the cost-rates along the edges are r_{xy} and r_{vw} respectively, (for example, two road segments where paths leave road $\overline{XY}$ from point Y or leave road $\overline{VW}$ at point W), the boundary between them is a segment of line L defined as follows. Let D_{xy} be the line which forms angle ψ_{xy} with line $\overline{XY}$, is distance c_y from point Y, and lies on the side of $\overline{XY}$ which does not include the region of which $\overline{XY}$ is the root. Let D_{vw} be the line which forms angle ψ_{vw} with line $\overline{VW}$, is distance c_w from point W, and lies on the side of $\overline{VW}$ which does not include the region of which $\overline{VW}$ is the root. Let P_0 be the point of intersection of D_{xy} and D_{vw}, and let α be the angle between line $\overline{XY}$ and line $\overline{VW}$. Then the boundary lies on line L, which is the line through point P_0 which lies at an angle $(\alpha + \psi_{vw} + \psi_{xy})/2$ with both D_{xy} and D_{vw}.

PROOF V-0.3:(See Figure 39) Consider the set of points P with two optimal paths, $OPL_1 = [Q_1, Y \mid OPL(Y)]$, and $OPL_2 = [Q_2, W \mid OPL(W)]$, where Q_1 and Q_2 are the points at which the paths first enter edges $\overline{XY}$ and $\overline{VW}$ respectively. The cost of OPL_1 is $r_0\, d(P,Q_1) + \sin(\psi_{xy} d(Q_1,Y) + c_y$ and the cost of OPL_2 is $r_0\, d(P,Q_2) + \sin\psi_{vw} d(Q_2,W) + c_w$. By Figure 39, these are the perpendicular distances of P from two lines D_{xy} and D_{vw}, defined as follows. D_{xy} is the line which forms angle ψ_{xy} with $\overline{XY}$, is distance c_y from point Y, and lies on the opposite side of $\overline{XY}$ from the region of which $\overline{XY}$ is the root. D_{vw} is the line which forms angle ψ_{vw} with $\overline{VW}$, is distance c_w from point W, and lies on the opposite side of $\overline{VW}$ from the region of which $\overline{VW}$ is the root. From analytic geometry, a set of points equidistant from two lines is a line. The point P_0, where D_{xy} and D_{vw} intersect, is distance zero from both lines, and so lies on line L which includes the boundary. By basic plane geometry, the line equidistant from two intersecting lines is the line which bisects the angle between them. The angle between D_{xy} and D_{vw} is $(\alpha + \psi_{xy} + \psi_{vw})$, so that line L forms angle $(\alpha + \psi_{xy} + \psi_{vw})/2$ with both D_{xy} and D_{vw}. ♦

THEOREM V-0.4: ***(Boundary between two regions having paths which cross two different edges.)*** Given goal point G and two adjacent homogeneous-behavior regions with cost-rate r_0, one region having Snell's-Law edge $\overline{VW}$ and the other having Snell's-Law edge $\overline{XY}$, where paths which cross $\overline{VW}$ go directly to point U at cost-rate r_{vw}, paths which cross $\overline{XY}$ go directly to point Z at cost-rate r_{xy}, and where the total cost from U to the goal is c_u and from Z to the goal is c_z, the boundary between them consists of points P such that the

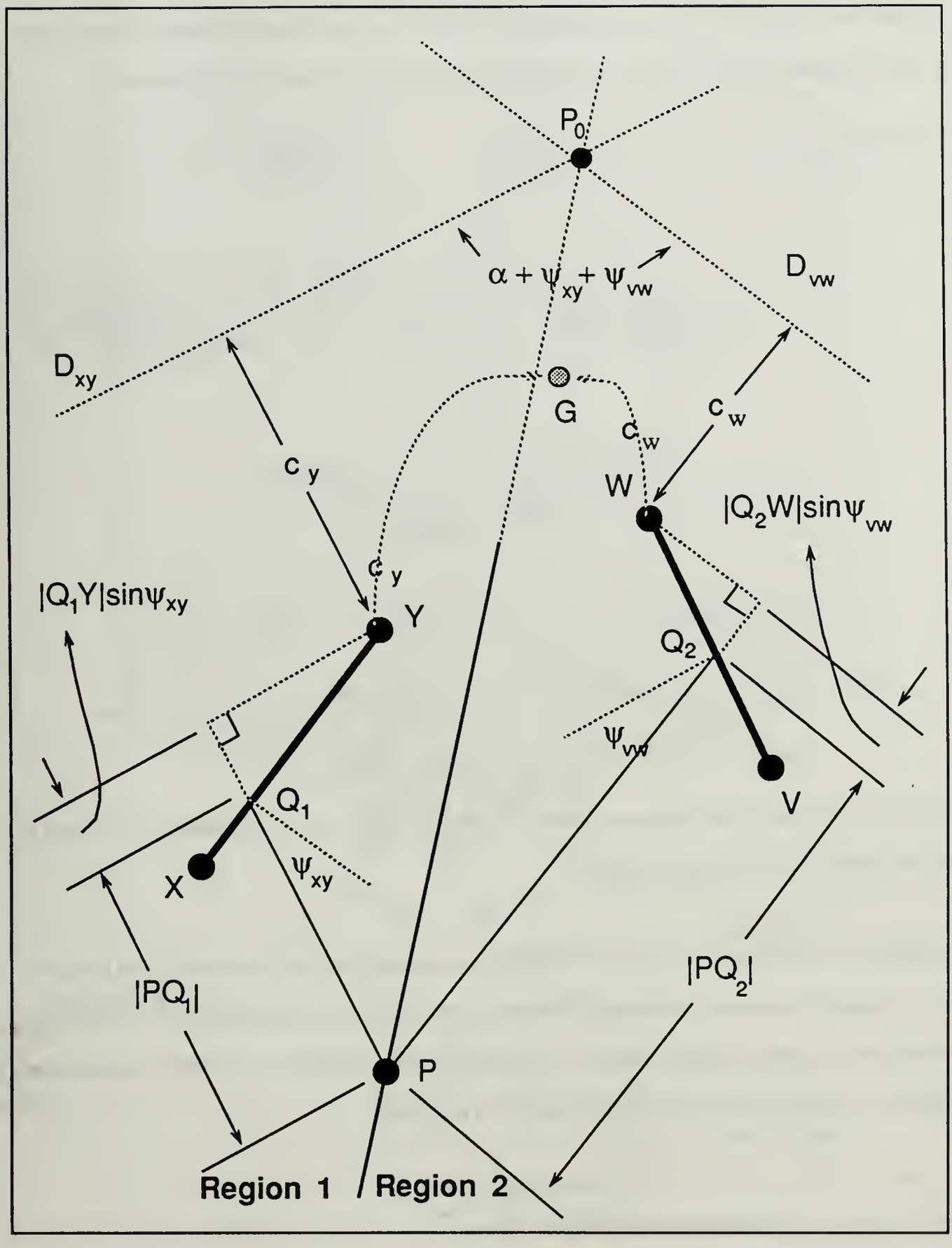


Figure 39
Boundary Between Homogeneous-Behavior Regions with Two Linearly-Traversed Roots

distance from P to edge $\overline{VW}$ is x_2, the distance from $\overline{VW}$ to U is x_1, the distance from point P to edge XY is y_2, and the distance from $\overline{XY}$ to Z is y_1, where the seven equations of Equation Set 4 are satisfied.

(Equation Set 4)

$$x_1 = \frac{d_1 \sin\gamma}{\cos\theta_1}, \qquad y_1 = \frac{d_0 \sin\beta}{\cos\theta_3},$$

$$y_2 = \frac{\sin\alpha}{\sin\theta_4}\left(\frac{d_1\cos(\theta_1 - \gamma)}{\cos\theta_1} - \frac{\cos(\theta_2 + \alpha)}{\cos\theta_2}\left(\frac{\frac{d_0\cos(\theta_3 - \beta)}{\cos\theta_2} - \frac{d_1\cos(\theta_1 - \gamma)\cos(\theta_4 + \alpha)}{\cos\theta_1\cos\theta_4}}{1 - \frac{\cos(\theta_2 + \alpha)\cos(\theta_4 + \alpha)}{\cos\theta_2\cos\theta_4}}\right)\right)$$

$$x_2 = \frac{\frac{d_0 \sin\alpha\cos(\theta_3 - \beta)}{\cos\theta_2\cos\theta_3} - \frac{d_1\cos(\theta_1 - \gamma)\cos(\theta_4 + \alpha)}{\cos\theta_1\cos\theta_4}}{\left(1 - \frac{(\cos(\theta_2 + \alpha)\cos(\theta_4 + \alpha)}{\cos\theta_2\cos\theta_4}\right)}$$

Boundary Condition: $r_{vw}x_1 + r_0x_2 = r_{xy}y_1 + r_0y_2$

Snell's Law for edge $\overline{VW}$: $r_{vw}\sin\theta_1 = r_0\sin\theta_2$

Snell's Law for edge $\overline{XY}$: $r_{xy}\sin\theta_3 = r_0\sin\theta_4$

where d_0, d_1, α, β, and γ are constants as shown in Figure 40, x_1, x_2, y_1, and y_2 are distances, and θ_1 and θ_3 are the dependent and independent variables.

PROOF V-0.4: (See Figure 40.) Given two adjacent regions with point P on their boundary, and given that the optimal paths from region 1 cross edge $\overline{VW}$ obeying Snell's Law, and then go through point U en route to the goal, and that optimal paths from region 2 cross edge $\overline{XY}$ obeying Snell's Law, and then go through point Z en route to the goal, with costs as shown, the boundary condition is

(4-1) $$r_{vw}x_1 + r_0x_2 = r_{xy}y_1 + r_0y_2.$$

The Snell's-Law conditions across edges $\overline{VW}$ and $\overline{XY}$ are

(4-2) $$r_{vw}\sin\theta_1 = r_0\sin\theta_2 \text{ and } r_{xy}\sin\theta_3 = r_0\sin\theta_4.$$

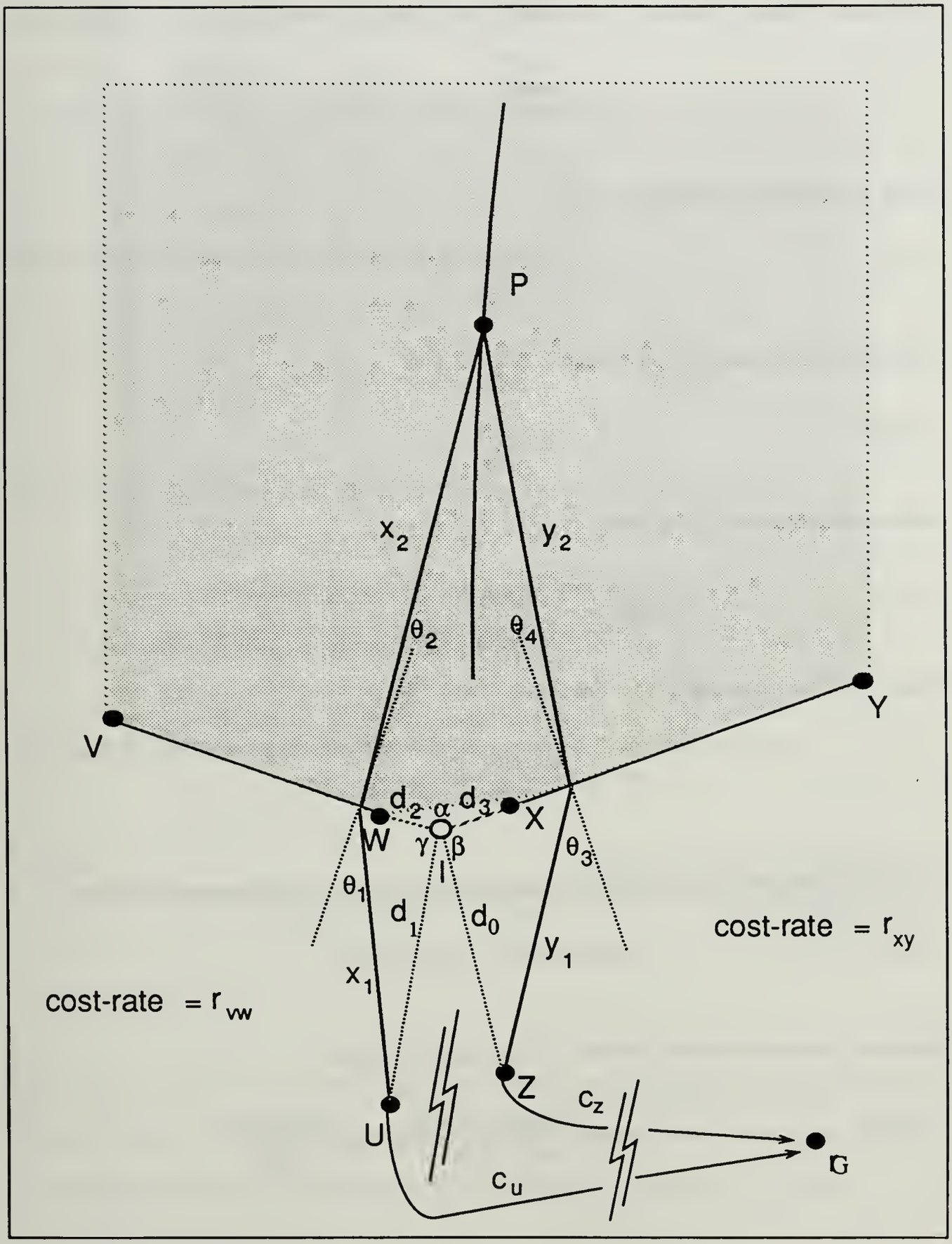


Figure 40
Boundary Between Homogeneous-Behavior Regions Each with a Snell's-Law Edge as Root

Applying trigonometric identities to the triangles UQ_1I and ZQ_2I gives the following for x_1 and y_1.

(4-3) $$x_1 = \frac{d_1 \sin\gamma}{\cos\theta_1},$$ (4-4) $$y_1 = \frac{d_0 \sin\beta}{\cos\theta_3}.$$

The law of sines applied to ΔUQ_1I gives that

(4-5) $$d_2 = \frac{x_1 \sin(\theta_1 - \gamma)}{\sin\gamma}.$$

Substituting the expression for x_1 in equation 4-3 into 4-5 gives

(4-6) $$d_2 = \frac{d_1 \cos(\theta_1 - \gamma)}{\cos\theta_1}.$$

The law of sines applied to ΔZQ_2I gives

(4-7) $$d_3 = \frac{y_1 \cos(\theta_3 - \beta)}{\sin\beta}.$$

Substituting for the expression for y_1 in Equation 4-4 into 4-7 gives

(4-8) $$d_3 = \frac{d_0 \cos(\theta_3 - \beta)}{\cos\theta_3}.$$

Applying trigonometric identities to the right triangle whose hypotenuse is the line segment $\overline{PQ_1}$ gives

(4-9) $$x_2\cos\theta_2 = d_3\sin\alpha - y_2\cos(\theta_4 + \alpha).$$

Substituting the expression for d_3 in Equation 4-8 into Equation 4-9 gives

(4-10) $$x_2 = \frac{d_0\cos(\theta_3 - \beta)\sin\alpha}{\cos\theta_3 \cos\theta_2} - y_2\frac{\cos(\theta_4 + \alpha)}{\cos\theta_2}.$$

Applying trigonometric identities to the right triangle whose hypotenuse is the line segment $\overline{PQ_1}$ gives

$$y_2\cos\theta_4 = d_2\sin\alpha - x_2\cos(\theta_2 + \alpha)\,. \tag{4-11}$$

Substituting the expression for d_2 in Equation 4-6 into Equation 4-11 gives

$$y_2 = \frac{d_1\cos(\theta_1 - \gamma)\sin\alpha}{\cos\theta_1\cos\theta_4} - x_2\frac{\cos(\theta_2 + \alpha)}{\cos\theta_4}\,. \tag{4-12}$$

Substituting the expression for y_2 in Equation 4-12 into Equation 4-10 and simplifying gives

$$x_2 = \frac{\dfrac{d_0\sin\alpha\cos(\theta_3 - \beta)}{\cos\theta_2\cos\theta_3} - \dfrac{d_1\cos(\theta_1 - \gamma)\cos(\theta_4 + \alpha)}{\cos\theta_1\cos\theta_4}}{\left(1 - \dfrac{\cos(\theta_2 + \alpha)\cos(\theta_4 + \alpha)}{\cos\theta_2\cos\theta_4}\right)} \tag{4-13}$$

Substituting the expression for x_2 in Equation 4-13 into Equation 4-12 and simplifying gives

$$y_2 = \frac{\sin\alpha}{\sin\theta_4}\left(\frac{d_1\cos(\theta_1 - \gamma)}{\cos\theta_1} - \frac{\cos(\theta_2 + \alpha)}{\cos\theta_2}\left(\frac{\dfrac{d_0\cos(\theta_3 - \beta)}{\cos\theta_2} - \dfrac{d_1\cos(\theta_1 - \gamma)\cos(\theta_4 + a)}{\cos\theta_1\cos\theta_4}}{1 - \dfrac{\cos(\theta_2 + \alpha)\cos(\theta_4 + \alpha)}{\cos\theta_2\cos\theta_4.}}\right)\right) \tag{4-14}$$

Equations 4-1, 4-2, 4-3, 4-4, 4-13, and 4-14 are exactly Equation Set 4. θ_1 and θ_3 must be iteratively set and the results of the first four equations checked in the boundary-condition equation, since there is no known closed form for Equation Set 4. The angles θ_2 and θ_4 are determined by the Snell's Law relations. ♦

THEOREM V-0.5: *(Boundary between a region with paths which go to and travel along a linearly-traversed edge ("road") and a region with paths which cross an edge)* Given goal point G and two adjacent homogeneous-behavior regions with cost-rate r_0, one region having linearly-traversed edge $\overline{VW}$ as root, and the other having as root Snell's-Law edge $\overline{XY}$, where $\overline{VW}$ is a sub-segment of some terrain-feature edge such that OPL(V) = [W | OPL(W)] and the cost-rate along the edge is r_{vw}, (for example, a road segment where paths leave the road from point W), and where paths which cross $\overline{XY}$ go directly to point Z at cost-rate r_{xy}, and where the total cost from W to the goal is c_w, and from Z to the goal is c_z. The boundary between the regions consists of points P such that the six equations of Equation Set 5 are satisfied.

(Equation Set 5) $$y_1 = \frac{d_3 \sin\beta}{\cos\theta_1} \qquad y_2 = x_1 \frac{\sin\alpha}{\cos\theta_2} - x_2 \frac{\cos(\psi+\alpha)}{\cos\theta_2} + d_2 \frac{\sin\gamma}{\cos\theta_2}$$

$$x_2 = \frac{d_2(\cos(\theta_2+\gamma)(r_0\sin\alpha - r_{vw}\cos\theta_2) + r_0\sin\gamma\cos(\theta_2-\alpha))}{\sin(\theta_2-\alpha-\psi)r_{vw}\cos\theta_2+r_0\cos(\theta_2-\alpha)(\sin(\theta_2-\alpha-\psi)+r_0(\cos\theta_2+\cos(\theta_2-\alpha)))}$$
$$+ \frac{d_3\cos\theta_2(r_0\sin\alpha\cos\beta+ r_{xy}\sin\beta\cos(\theta_2-\alpha)) + (c_z - c_w)\cos\theta_2\cos(\psi-\alpha)}{\sin(\theta_2-\alpha-\psi)r_{vw}\cos\theta_2+r_0\cos(\theta_2-\alpha)(\sin(\theta_2-\alpha-\psi)+r_0(\cos\theta_2+\cos(\theta_2-\alpha)))}$$

$$x_1 = \frac{(\frac{r_0\sin\alpha}{\cos(\theta_2-\alpha)} + r_0(\frac{\cos\theta_2+\cos(\psi+\alpha)}{\sin(\theta_2-\alpha-\psi)} - 1))\,(d_2\cos(\theta_2+\gamma)+d_3\cos\beta\cos\theta_2)}{r_{vw}\cos\theta_2 + r_0\cos(\theta_2-\alpha)(\frac{\cos\theta_2+\cos(\psi+\alpha)}{\sin(\theta_2-\alpha-\psi)} - 1)}$$
$$+ \frac{d_2r_0\sin\gamma + d_3(\frac{r_{vw}\cos\beta\cos^2\theta_2}{\cos(\theta_2-\alpha)} + r_{xy}\sin\beta\cos\theta_2) + (c_z - c_w)\cos\theta_2}{r_{vw}\cos\theta_2 + r_0\cos(\theta_2-\alpha)(\frac{\cos\theta_2+\cos(\psi+\alpha)}{\sin(\theta_2-\alpha-\psi)} - 1)}$$

Snell's Law condition for edge $\overline{XY}$: $r_{xy}\sin\theta_1 = r_0\sin\theta_2$

Snell's Law condition for edge $\overline{VW}$: $\sin\psi = r_{vw} / r_0$

PROOF V-0.5:(See Figure 41.) Given two adjacent regions with point P on their boundary, and given that the optimal paths from region 1 go directly to edge $\overline{VW}$ and travel along it to point W, and that optimal paths from region 2 cross edge $\overline{XY}$ obeying Snell's Law, and then go through point Z en route to the goal, with costs as shown in Figure 41, the boundary condition is

$$c_r y_2 + r_{xy} y_1 + c_z = c_r x_2 + r_{vw} x_1 + c_w .$$

At the two edges, the Snell's-Law conditions are

$$c_v \sin\theta_2 = r_{xy} \sin\theta_1$$

$$\text{and} \quad \sin\psi = r_{vw} / c_r .$$

The same type of trigonometric and algebraic reasoning used in Proof V-0.4 leads to the equations listed in Equation Set 5. Since there is no closed-form expression for the boundary, an approximation is computed using a finite number of points. The procedure for plotting a point on the boundary is to set θ_1, use the first Snell's-Law condition to solve for θ_2, and then solve for x_1 and x_2. ♦

THEOREM V-0.6: (*Boundary between two regions each having paths which cross two edges)* Given goal point G and two adjacent homogeneous-behavior regions with cost-rate r_0, one region having Snell's-Law edge $\overline{VW}$ and the other having Snell's-Law edge $\overline{RS}$, where paths which cross $\overline{VW}$ go from there at cost-rate r_{vw} directly to a Snell's-Law crossing at edge $\overline{XY}$, and then go at cost-rate r_{xy} directly to point Z_1; paths which cross $\overline{RS}$ go from there at cost-rate r_{rs} directly to a Snell's-Law crossing at edge $\overline{TU}$, and then go at cost-rate r_{tu} directly to point Z_2, and where total cost from Z_1 to the goal is c_1 and from Z_2 to the goal is c_2, the boundary between them consists of points P such that the path distance from P to edge $\overline{VW}$ is y_3, the path distance from $\overline{VW}$ to $\overline{XY}$ is y_2, the path distance from $\overline{XY}$ to point P is y_1, the path distance from point P to edge $\overline{RS}$ is x_3, the path distance from $\overline{RS}$ to $\overline{TU}$ is x_2, and the path distance from $\overline{TU}$ to Z_2 is x_1, where the fourteen equations of Equation Set 6 are satisfied.

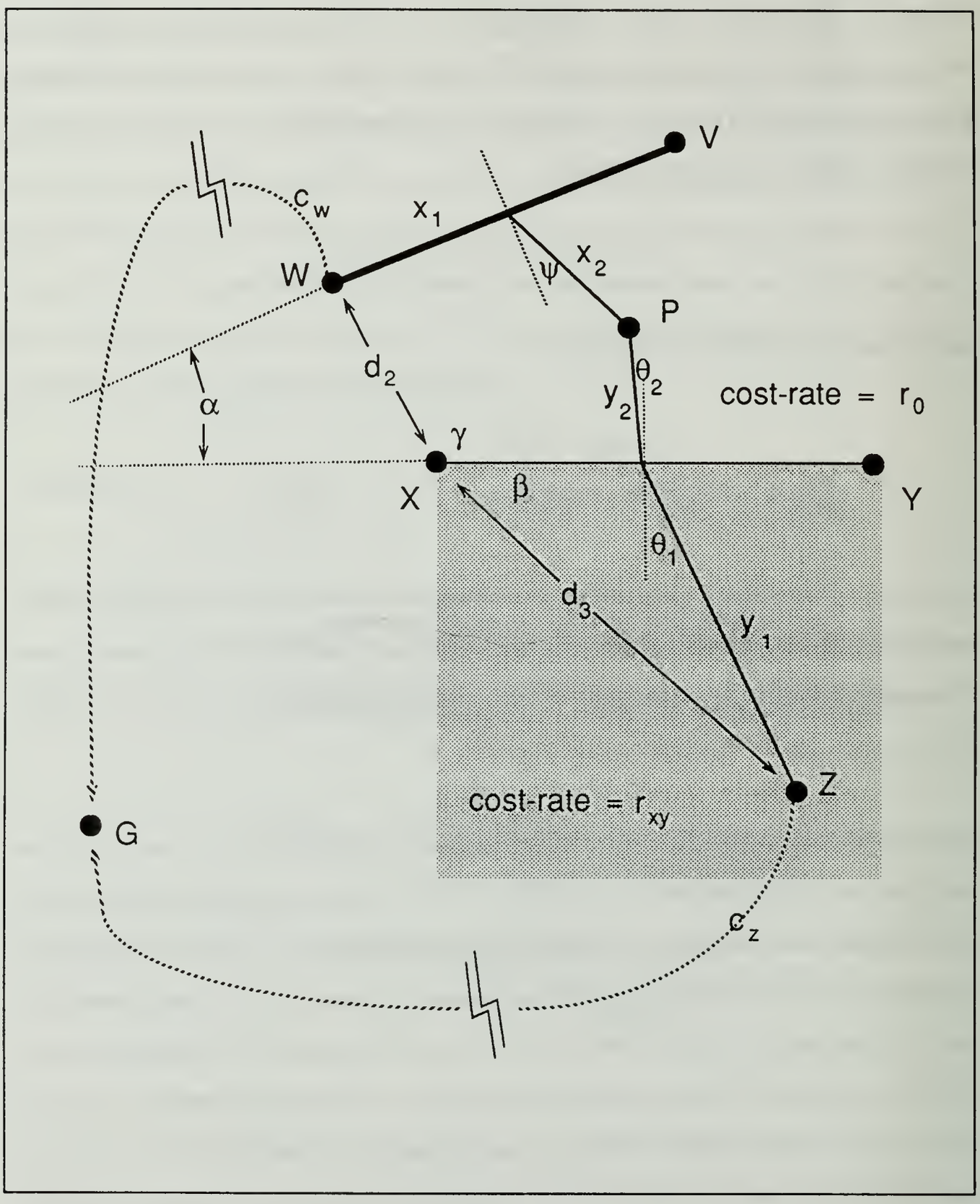


Figure 41

Boundary Between Homogeneous-Behavior Regions with a Snell's-Law Edge and a Linearly-Traversed Edge as Roots

(Equation Set 6) $x_1 = \frac{d_7 \sin\beta_1}{\cos\theta_8}$, $y_1 = \frac{d_5 \sin\beta_1}{\cos\theta_1}$,

$$x_2 = \frac{\sin\alpha_3}{\cos\theta_6}\left(d_4 - \frac{d_7\cos(\beta_2 - \theta_8)}{\cos\theta_8}\right), \quad y_2 = \frac{\sin\alpha_1}{\cos\theta_3}\left(d_1 - \frac{d_5\cos(\beta_1 - \theta_1)}{\cos\theta_1}\right),$$

$$x_3 = -\frac{\sin(\alpha_3 + \gamma_1)}{\cos(\theta_5 + \alpha_4 + \gamma_1)}\left(d_3 - \frac{d_4\cos\theta_7}{\sin\theta_6} + \frac{d_7\cos(\beta_2 - \theta_8)\cos\theta_7}{\cos\theta_8}\right),$$

$$y_3 = \frac{\sin\gamma_1}{\cos(\theta_4 - \gamma_1)}\left(d_2 - \frac{d_1\cos\theta_2}{\cos\theta_3} + \frac{d_5\cos(\beta_1 - \theta_1)\cos\theta_2}{\cos\theta_1}\right)$$

$$\frac{\cos(\theta 4 - \gamma 1)}{\cos(\theta_5 + \alpha_4 + \gamma_1)} = \frac{d_3 - \frac{d_4\cos\theta_7}{\sin\theta_6} + \frac{d_5\cos(\beta_2 - \theta_8)\cos\theta_4}{\cos\theta_8}}{d_3 - \frac{d_4\cos\theta_7}{\sin\theta_6} + \frac{d_7\cos(\beta_2 - \theta_8)\cos\theta_7}{\cos\theta_8}}$$

Boundary Condition: $r_0y_3 + r_{vw}y_2 + r_{xy}y_1 = r_0x_3 + r_{rs}x_2 + r_{tu}x_1$

Snell's Law: $r_{vw}\sin\theta_2 = r_{xy}\sin\theta_1$ $r_0\sin\theta_4 = r_{vw}\sin\theta_3$

$r_{rs}\sin\theta_7 = r_{tu}\sin\theta_8$ $r_0\sin\theta_5 = r_{rs}\sin\theta_6$

Trigonometric Identities: $\theta_3 = \alpha_1 - \theta_2$ $\theta_6 = \alpha_3 - \theta_7$

PROOF V-0.6: (See Figure 42.) Given two adjacent regions with point P on their boundary, and given that the optimal paths from region 1 cross edge $\overline{VW}$ obeying Snell's Law, then go straight to edge $\overline{XY}$ and cross it obeying Snell's Law, and then go through point Z_1 en route to the goal, and that optimal paths from region 2 cross edge $\overline{RS}$ obeying Snell's Law, then go straight to edge $\overline{TU}$ and cross it obeying Snell's Law, and then go through point Z_2 en route to the goal, with costs as shown in Figure 42, the boundary condition is

$$r_0y_3 + r_{vw}y_2 + r_{xy}y_1 = r_0x_3 + r_{rs}x_2 + r_{tu}x_1 .$$

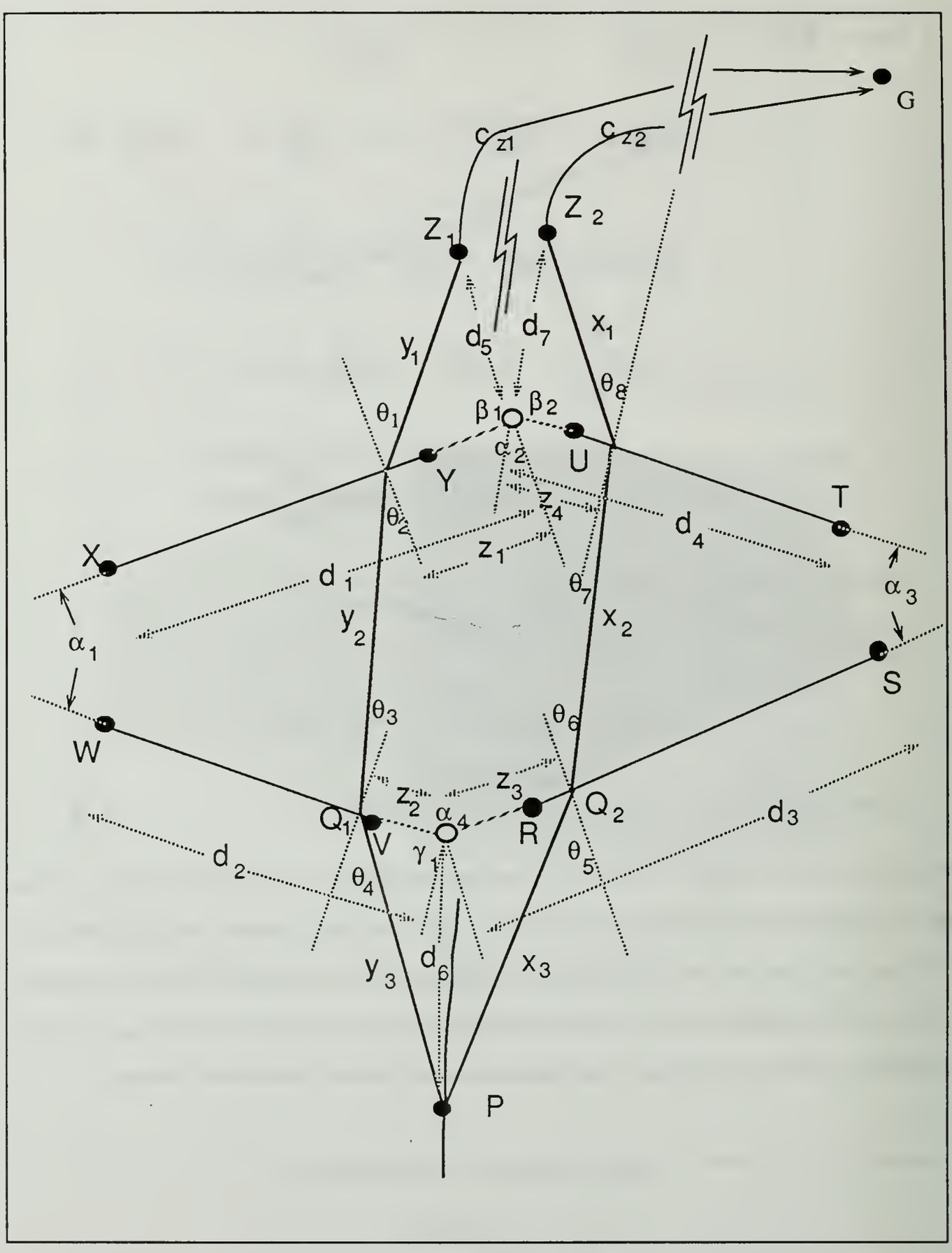


Figure 42
Boundary Between Homogeneous-Behavior Regions each with Two Snell's-Law Edges as Roots

At each edge, the Snell's-Law conditions are

$$r_{vw}\sin\theta_2 = r_{xy}\sin\theta_1$$

$$r_0\sin\theta_4 = r_{vw}\sin\theta_3$$

$$r_{rs}\sin\theta_7 = r_{tu}\sin\theta_8$$

$$r_0\sin\theta_5 = r_{rs}\sin\theta_6 .$$

Trigonometric identities applied to $\Delta V_1P_1P_2$ and $\Delta V_2P_3P_4$ give the relations

$$\theta_3 = \alpha_1 - \theta_2 \quad \text{and}$$

$$\theta_6 = \alpha_3 - \theta_7 \ .$$

Applying to the diagram of Figure 42 the same type of trigonometric and algebraic reasoning used in Proof V-0.4 leads to the equations listed in Equation Set 6. By solving for θ_1 and θ_8, a point P on the boundary can be found. Since there is no closed-form expression for the boundary, an approximation is used where a finite number of points are plotted. Since there is no closed-form expression for θ_8 as a function of θ_1, the procedure for plotting a point on the boundary is to set θ_1, iteratively search for a value of θ_8 for which the equations of Equation Set 6 are satisfied (within some allowable error), and then trace the Snell's-Law path according to the heading for θ_1 using the values for y_1, y_2, and y_3, or according to the heading for θ_8 using the values for x_1, x_2, and x_3. Note also that the expression for γ_1 is not in closed form, and so must be found by iterative means. ♦

E. PROOFS FOR BOUNDARIES ASSOCIATED WITH PRIMITIVE TERRAIN-FEATURE TYPES

LEMMA V-1.1: If there are feasible paths from a vertex of a polygonal obstacle, then the *obstacle edges* constitute boundaries between homogeneous-behavior regions.

PROOF V-1.1: Trivially true. ♦

LEMMA V-1.2: Each vertex V of an obstacle hidden edge generates a linear *shadow boundary* which is the ray lying on the line defined by V and the first point P on OPL(V), starting at V and lying in the opposite direction from P.

PROOF V-1.2: Note that if V joins a hidden edge and a visible edge, point P will not be on the obstacle perimeter by the definition of a visible edge; if V joins two hidden edges, P will be the other vertex of one of the hidden edges. We prove first that there is a single shadow boundary associated with each hidden-edge vertex, second that no vertices other than hidden-edge vertices generate shadow boundaries, and third, that the shadow boundary is a ray defined as stated in Lemma V-1.2.

First, consider point Q near V, a hidden-edge vertex. Let P be the first point on OPL(V). Then one of three cases holds (see Figure 43): either (a) Q_a is in the obstacle interior, or (b) Q_b and P are intervisible, or (c) Q_c and P are not intervisible. Clearly, if V joins a hidden and a visible edge, Figure 43a applies, and if V joins two hidden edges, Figure 43b applies. Q_a is separated from Q_b and Q_c, not by shadow boundaries, but by obstacle-edge boundaries. The optimal path from Q_b is $(Q_bG)^* = ((Q_bP)^* (PG)^*)$, where $(Q_bP)^*$ is the line segment $\overline{PQ_b}$. Thus the optimal-path list from Q_b is $OPL(Q_b) = [P \mid OPL(P)]$. The optimal path from Q_c is $OPL(Q_cG)^* = ((Q_cV)^*(VP)^*(PG)^*)$, where $(Q_cV)^*$ and $(VP)^*$ are the line segments $\overline{VQ_c}$ and $\overline{VP}$ respectively. Thus the optimal-path list from Q_c is $OPL(Q_c) = [V, P \mid OPL(P)]$. Thus $OPL(Q_b) \neq OPL(Q_b)$, so Q_b and Q_c are in different regions, so there is a boundary between them.

We show secondly that no other vertices generate shadow boundaries. Assume vertex V does not join a hidden and a visible edge, or two hidden edges. Then it joins two visible edges. Thus, OPL(V) does not in-

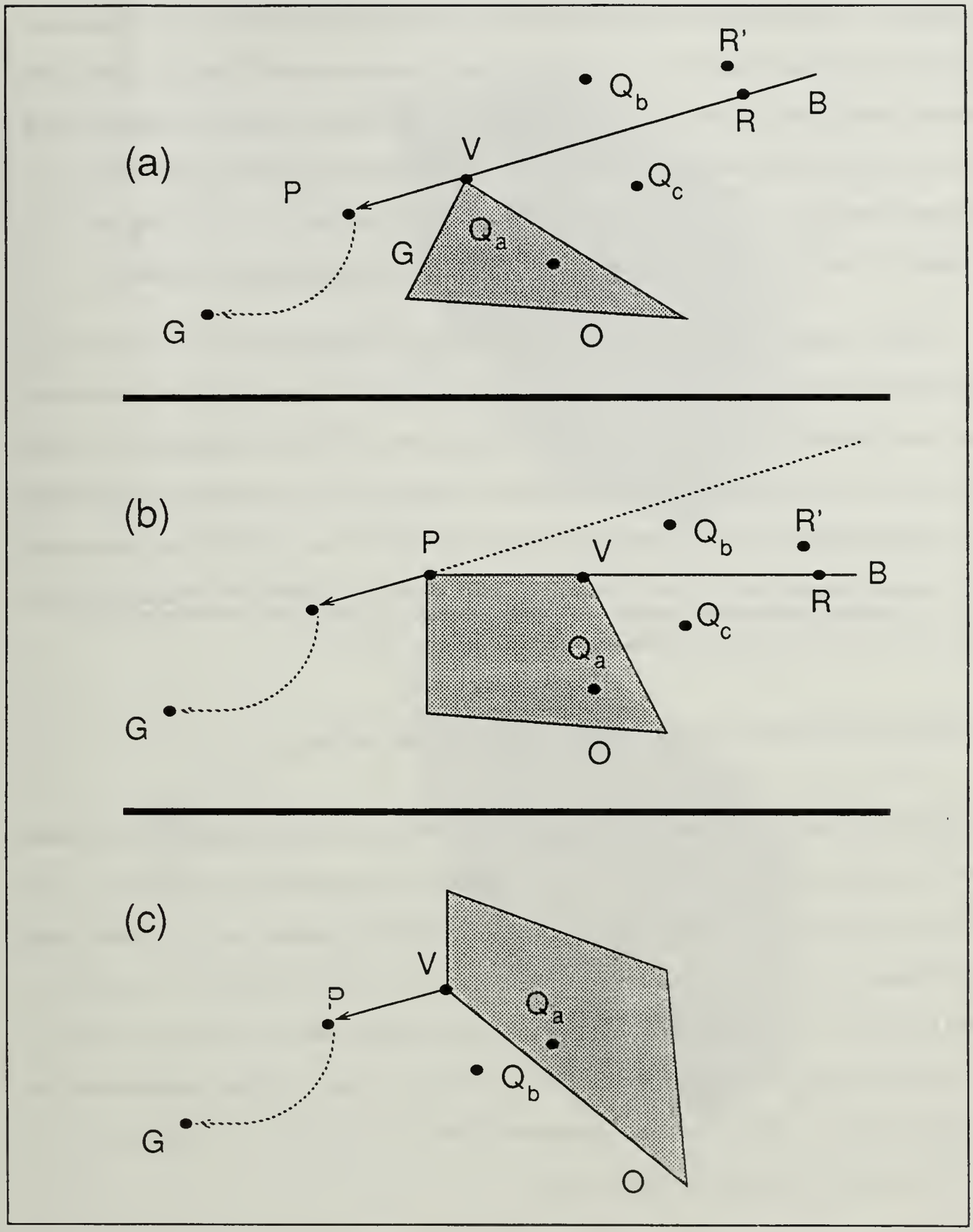


Figure 43
Obstacle Shadow Boundaries

clude any vertices of the obstacle, from the definition of a visible edge. By Assumption I-3, V is not on a non-trivial (i.e., non-obstacle-edge) boundary. Consider a point Q arbitrarily close to V. Clearly, Q is either in the obstacle interior (call it Q_a) or in its exterior (including its edges) (call it Q_b). Clearly, Q_a is separated from Q_b by an obstacle-edge boundary. Now in the absence of externally-generated boundaries in the vicinity of V_1, Q_b can be made close enough to V that it is in the same region as V, and so OPL(V) = OPL(Q_b). Thus in the vicinity of V, there is only one exterior region, and so V does not generate any shadow boundaries.

Thirdly, we show that each shadow boundary is a ray lying on the line defined by vertex V and P, the first point on OPL(V), starting at V and lying away from P. Consider a point R on ray B in Figure 43a or 43b. By convention, let points on B not be intervisible with P. Then $(RP)^* = (\overline{RV}\ \overline{VP})$. Now consider R' arbitrarily close to R but intervisible with P. By the definition of intervisibility, $(R'P)^*$ is a straight-line segment. Since R' is arbitrarily close to P, $(R'P)^*$ must be arbitrarily close to $(\overline{RV}\ \overline{VP})$, and so $(\overline{RV}\ \overline{VP})$ must be a straight-line segment, collinear with P, V, and R. Since B separates the region with OPL = [V, P | OPL(P)] from the region with OPL = [P | OPL(P)], B must begin at V and lie away from P. ♦

LEMMA V-1.3: A convex polygonal obstacle has exactly one opposite edge.

PROOF V-1.3: First, we show that obstacle O with n distinct vertices has at least one opposite edge. Assume O in Figure 44a has no opposite edge. Then for any hidden edge V_iV_{i+1}, either $OPL(V_i) \subset OPL(V_{i+1})$, or $OPL(V_{i+1}) \subset OPL(V_i)$, or else V_iV_{i+1} would be an opposite edge. Now consider vertex V_1, a vertex joining a hidden and a visible edge. By the definition of visible edges, $\forall V_k \in O$, $V_k \notin OPL(V_1)$. Therefore, it must be that $OPL(V_2) \not\subset OPL(V_1)$. Since V_1V_2 is not an opposite edge, $OPL(V_1) \subset OPL(V_2)$. Then by induction on i, similar reasoning shows that $\forall i$, $OPL(V_i) \subset OPL(V_{i+1})$. For i = n, similar reasoning shows that $OPL(V_n) \subset OPL(V_1)$. But this statement contradicts that $\forall V_k \in O$, $V_k \notin OPL(V_1)$. Therefore by contradiction, obstacle O has at least one opposite edge.

Now assume that there are n distinct opposite edges, where $n \geq 2$. Choose any two opposite edges, say V_iV_{i+1} and V_jV_{j+1}, and without loss of generality assume that $j \geq i+1$ as shown in Figure 44b. Since V_iV_{i+1}

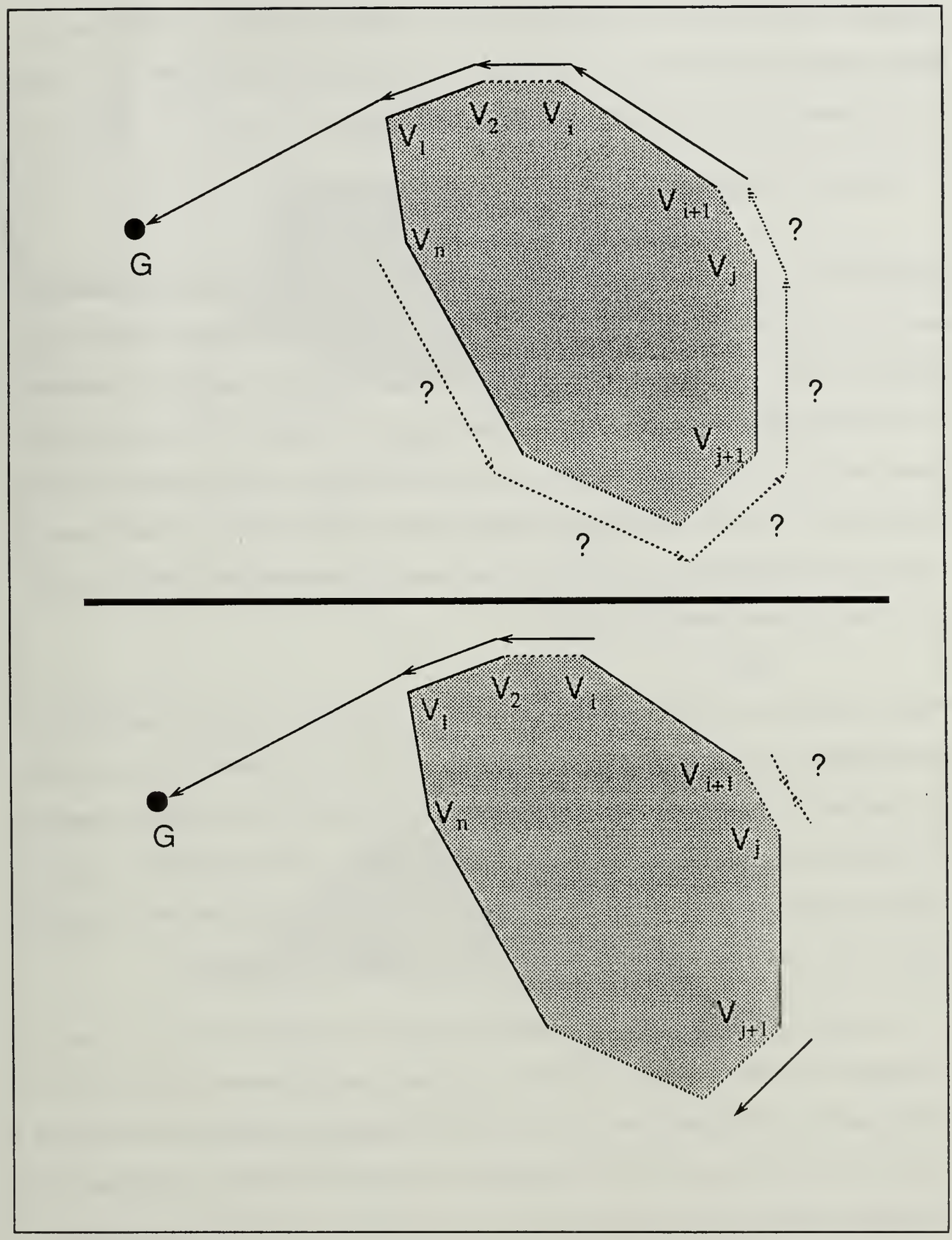


Figure 44
Uniqueness of Obstacle Opposite Edge

is an opposite edge, $V_i \notin OPL(V_{i+1})$. Therefore $OPL(V_{i+1}) \subset OPL(V_j)$ and $OPL(V_{i+1}) \subset OPL(V_{j+1})$. By similar reasoning with respect to edge V_jV_{j+1}, $OPL(V_{j+1}) \subset OPL(V_i)$ and $OPL(V_{j+1}) \subset OPL(V_{i+1})$. But this is a contradiction, so there must be no more than one opposite edge.

Therefore a convex polygonal obstacle has exactly one opposite edge. ♦

LEMMA V-1.4: An *opposite-edge boundary* emanates from each obstacle opposite edge and consists of segments of hyperbolas such that an initial hyperbola segment starts at the opposite point and is defined by considering the vertices V_1 and V_2 of the opposite edge as its foci, with hyperbolic constant being the absolute value of the difference of the costs of $(V_1G)^*$ and $(V_2G)^*$, as specified in Equation 1. If at any point a shadow boundary intersects the opposite-edge boundary, it will continue along a new hyperbola segment defined by considering as foci, first, the vertex of the edge which generated the shadow boundary and which is closer to the goal of the two vertices of that edge, and second, the focus of the previous hyperbola which is not also a vertex of the edge which generated the shadow.

PROOF V-1.4: Given a convex polygonal obstacle O with opposite edge V_1V_2, and given point X on V_1V_2 such that $\exists (XG)_1^*$ and $(XG)_2^*$, $(XG)_1^* \neq (XG)_2^*$, i.e., X is the opposite point. Since V_1V_2 is a hidden edge, then it must be that $OPL_1(X) = [V_1 \mid OPL(V_1)]$ and $OPL_2(X) = [V_2 \mid OPL(V_2)]$ (see Figure 45a). Consider point P arbitrarily close to X in the obstacle exterior. By Theorem I-2, $(PV_1)^* = |PV_1|$ and $(PV_2)^* = |PV_2|$, because no other terrain features intervene, so P is in both the homogeneous-behavior region with V_1 as root and the region with V_2 as root. By Theorem V-0.1, the set of points P is described by Equation 1.

Let B_1 be the set of points over which P obeys the Equation 1. As P moves away from X, it lies on B_1 only as long as $PV_1 \subset (PG)_1^*$ and $PV_2 \subset (PG)_2^*$, i.e., as long as the line segment from P to both vertices are part of the respective optimal paths from P in the two directions. If at some point Z it becomes true that $PV_i \not\subset (PG)_i^*$, for i=1 or i=2, then at that point B_1 must have intersected shadow boundary i (see Figure 45b). Now the same reasoning as above applies to the point V_k, where $OPL(V_i) = [V_k \mid OPL(V_k)]$, and so another hyperbola branch B_2 becomes the adjoining portion of the opposite-edge boundary. Since point Z lay on both hyperbola branches B_1 and B_2, it must be that B_1 and B_2 intersect at point Z. The same reasoning continues

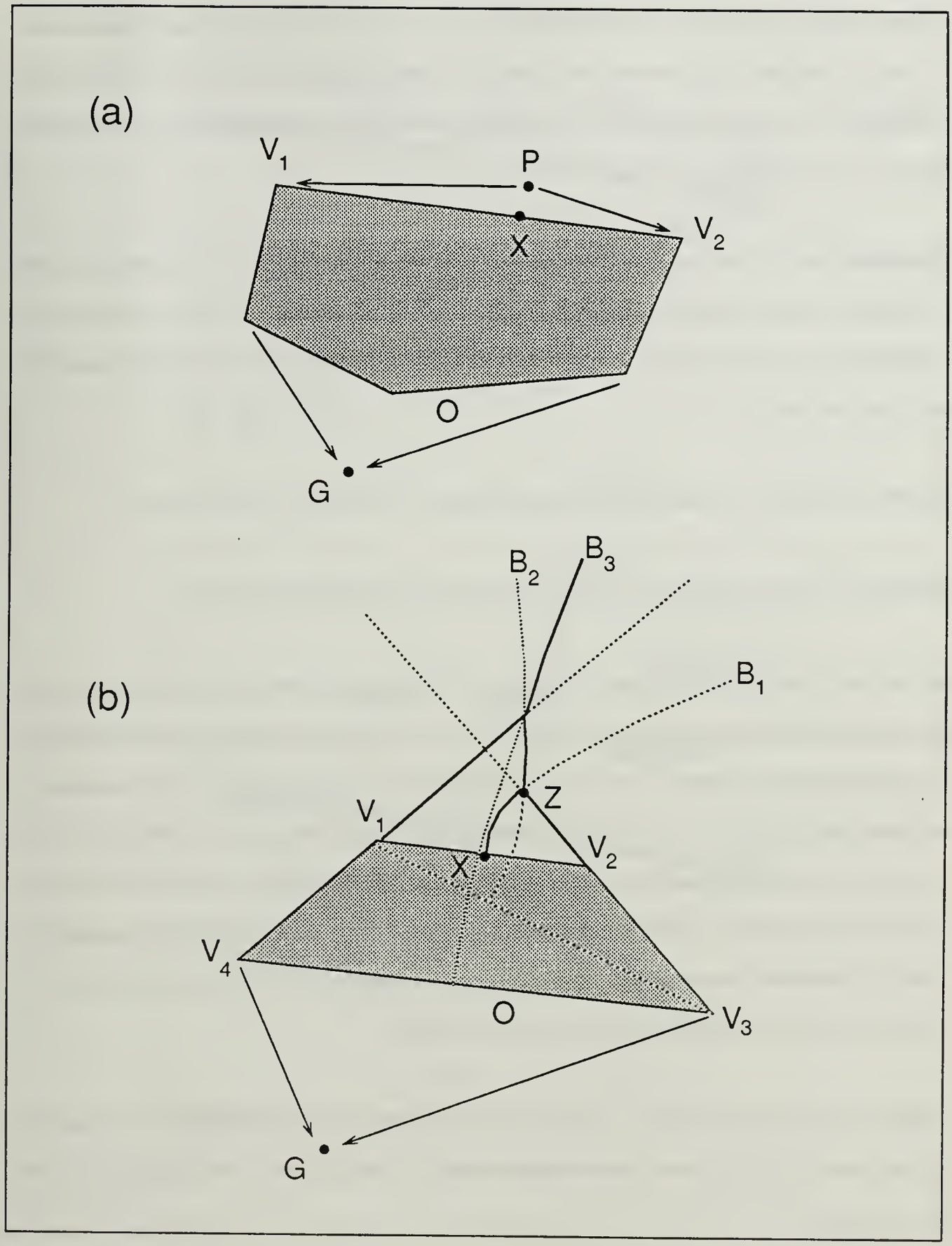


Figure 45
Obstacle Opposite-Edge Boundary

to apply as long as B_j intersects any shadow boundary of obstacle O. Therefore the opposite-edge boundary is a connected sequence of hyperbola segments starting at the opposite point, and for each segment consisting of a portion of the hyperbola branch with the two visible obstacle vertices as foci and the hyperbolic constant being $c_2 - c_1$, where $c_2 > c_1$. ♦

THEOREM V-1: A convex polygonal obstacle in homogeneous background terrain with specified goal-point will generate as boundaries the obstacle edges, shadow boundaries from each vertex of a hidden edge as specified in Lemma V-1.2, and a single opposite-edge boundary consisting of piecewise hyperbolic segments as specified in Lemma V-1.4.

PROOF V-1: Theorem V-1 follows directly from Lemma V-1.1, Lemma V-1.2, and Lemma V-1.4.

LEMMA V-2.1: A river segment, or *river-edge*, constitutes a boundary between regions.

PROOF V-2.1: (See Figure 46a.) Given river segment V_1V_2, and point X_1 arbitrarily close to V_1V_2 having optimal-path list $OPL(X_1) = [W \mid OPL(W)]$ where $W \notin V_1V_2$, i.e., X_1's optimal path does not cross the river, and point X_2 arbitrarily close to V_1V_2 on the opposite side V_1V_2. Now X_2 may have one of three possible optimal-path lists: $OPL_a(X_2) = [V_1 \mid OPL(V_1)]$ i.e., it goes around end 1 of the river, or $OPL_b(X_2) = [V_2 \mid OPL(V_2)]$, i.e., it goes around end 2 of the river, or $OPL_c(X_2) = [[V_1V_2] \mid OPL(W)]$ where $[V_1V_2]$ specifies that the path crosses the river without changing direction, and W is the next point on the optimal-path list. Since in all three cases, the optimal-path list of X_2 is different from that of X_1, therefore X_1 and X_2 are in different regions. Therefore the river edge constitutes a boundary. ♦

LEMMA V-2.2: Each river vertex V with $OPL(V) = [W \mid OPL(W)]$ which is an endpoint of a river segment not joining any others will generate a *shadow boundary* which is a ray lying on the line VW, starting at V and lying away from W.

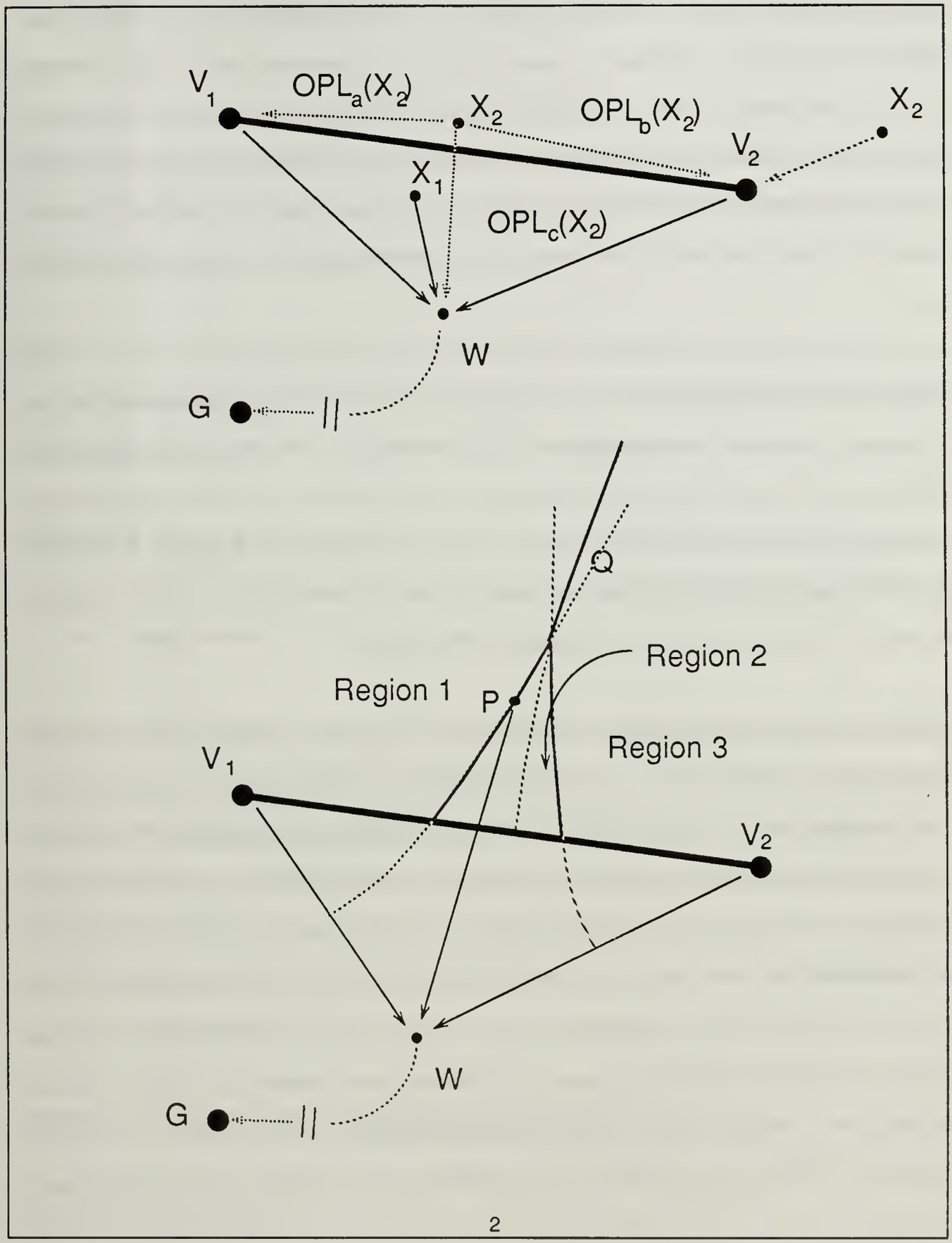


Figure 46
River Segment Boundaries

PROOF V-2.2: Given the same situation as in Proof V-2.1, analyze $OPL_a(X_2) = [V_1 \mid OPL(V_1)]$ and $OPL_b(X_2) = [V_2 \mid OPL(V_2)]$, with respect to vertices V_1 and V_2 in the same manner as in Proof V-1.2 to show that there are rays emanating from V_1 and V_2 lying away from the goal which act as boundaries between optimal paths which go around the vertices and those which bypass them. Note that, assuming a positive river-crossing cost, location c for X_2 will never be such that X_2, V_i, and the next point in $OPL_c(V_i)$ are collinear, because if so, it will be less costly for the optimal path to avoid crossing the river and go around vertex V_i instead. ♦

LEMMA V-2.3: A river segment with vertex V with $OPL(V) = [W \mid OPL(W)]$ not adjoining any other river segment may have a *river-crossing boundary* which is a segment of one branch of a hyperbola constructed by considering as foci the points V and W, with hyperbolic constant $c = |VW| - c_r$, where c_r is the fixed river-crossing cost. This boundary will exist if the branch closer to V intersects the river segment. The boundary consists of the portion of the hyperbola branch between the intersection of the branch with the river, and the first point of intersection of the branch with another river boundary.

PROOF V-2.3: Consider point P which lies in the shadow of river segment with vertex V as in Figure 46b, where r_0 is the cost rate for travel in the background region. As in Figure 46a, there are only three possible ways the optimal path from P can go initially. If P lies on a boundary between paths which cross the river paying the fixed crossing cost, and paths which go through V, the first region has V as its root and the second region has the river segment as its root. Optimal paths crossing river segments do not change headings. Therefore, the path from P to W has cost $c_{pw} = r_0\, d(P,W) + c_r$. The cost of the path from P to V has cost $c_{pv} = r_0\, d(P,V)$, as usual with a point root, and the cost c_w from W is known. But this is just as if paths in region 2 had W as a root, where the cost from W to G was $c_w + c_r$. Thus, the boundary separates two regions whose roots are points, so by Theorem V-0.1, the boundary is a hyperbola segment described by Equation 1. If c_r and the orientation of $\overline{VW}$ are such that the boundary does not intersect the river segment between V and U, it must

be that for all points in the shadow of the river segment, it is more costly to cross the river than to go around via V. ♦

LEMMA V-2.4: A river segment with vertices V_1 and V_2 has an *opposite-edge boundary* which lies on the hyperbola formed by considering each vertex as a focus conforming to Equation 1, and lies on the branch of the hyperbola which is closer to the vertex with higher-cost optimal path.

PROOF V-2.4: Consider point Q in Figure 46b. This point is on the boundary which separates region 1 from region 3. Optimal path from Q through region 1 goes through V, while the optimal path through region 2 goes through U. Thus, the boundary separates regions whose roots are both points, so Theorem V-0.1 applies. ♦

THEOREM V-2: An isolated river segment has a river-edge boundary, two shadow boundaries formed as specified in Lemma V-2.2, an opposite-edge boundary formed as specified in Lemma V-2.4, and either two, or no, river-crossing boundaries as specified in Lemma V-2.3.

PROOF V-2: Consider points X_1, X_2, V_1 and V_2 as in Proof V-2.1 and Figure 46a, with optimal-path lists $OPL(X_1)$, $OPL(X_a)$, $OPL(X_b)$, and $OPL(X_c)$ as described in Proof V-2.1. Clearly, these four optimal-path lists are the only ones possible for points arbitrarily close to an isolated river segment, so by the definition of a homogeneous-behavior region, there are no more than four regions associated with a river segment. Thus the only boundaries possible adjacent tc an isolated river segment are those between pairs of these four regions, plus a fifth, the region unaffected by the river. The form of each boundary follows directly from Lemmas V-2.1, V-2.2, V-2.3, and V-2.4. ♦

LEMMA V-3.1: A road-edge forms a boundary between homogeneous- behavior regions.

PROOF V-3.1: Trivially true. ♦

LEMMA V-3.2: Given road segment with goal G, one vertex V, and the other vertex's location unspecified, and cost-rate r_r, with cost-rate in the background r_0. If the characteristic wedge ∇AGB as defined in Chapter V lies "inside" road- vertex V, two *road-end/road-travelling* boundaries will be formed as rays with vertex at V, each lying so that its angle with the road is $\pi/2 + \psi$.

PROOF V-3.2: Consider the road segment of Figure 47a, with goal G, one vertex V, and the other vertex's location unspecified, and cost-rate r_r, with cost-rate in the background r_0. As shown in [Ref. 2], paths will enter leave a road interior only at the critical angle $\psi = \sin^{-1}(r_r/r_0)$. Thus a path leaving the road to point G will do so at point A. If $\overline{GA}$ does not intersect the road at or to the "left" (in the figure) of V, no paths will travel along the road from the direction of V. Otherwise, ∇AGB is said to lie "outside" V, and paths travel along the road from V. Consider points P_1 and P_2 in the vicinity of P. If P is arbitrarily close to V, the path from P_1 will enter the road at angle ψ en route to A, while the path from P_2 will enter the road at V. Thus, the set of boundary points P lies on ray $\overline{VP}$ such that $\angle PVA = \pi/2 + \psi$. The same reasoning with respect to point Q gives that ray VQ also is a boundary.♦

LEMMA V-3.3: Given road segment with goal G, one vertex V, and the other vertex's location unspecified, and cost-rate r_r, with cost-rate in the background r_0. If the characteristic wedge ∇AGB lies "inside" road-vertex V, a *road-end/goal* boundary will exist on the V end of the road segment, forming a segment of a hyperbola with V and G as foci and obeying Equation 1.

PROOF V-3.3: Consider point P in Figure 47b, with $OPL_1(P) = [V, A, G]$, and $OPL_2(P) = [G]$. Since the two regions through which the optimal paths from P lie have points as roots, Equation 1 applies, and the boundary is a hyperbola segment with V and G as foci. The boundary will begin at the point at which the hyperbola intersects the road- end/road-travelling boundary of Lemma V-3.2.♦

LEMMA V-3.4: Given road segment with goal G, vertices V_1 and V_2, and cost-rate r_r, with cost-rate in the background r_0. If the characteristic wedge ∇AGB lies "inside" road-vertex V_1, a *near-side-road-travelling/goal* wil exist on the near side of the river which forms a parabola with focus G and directrix as specified

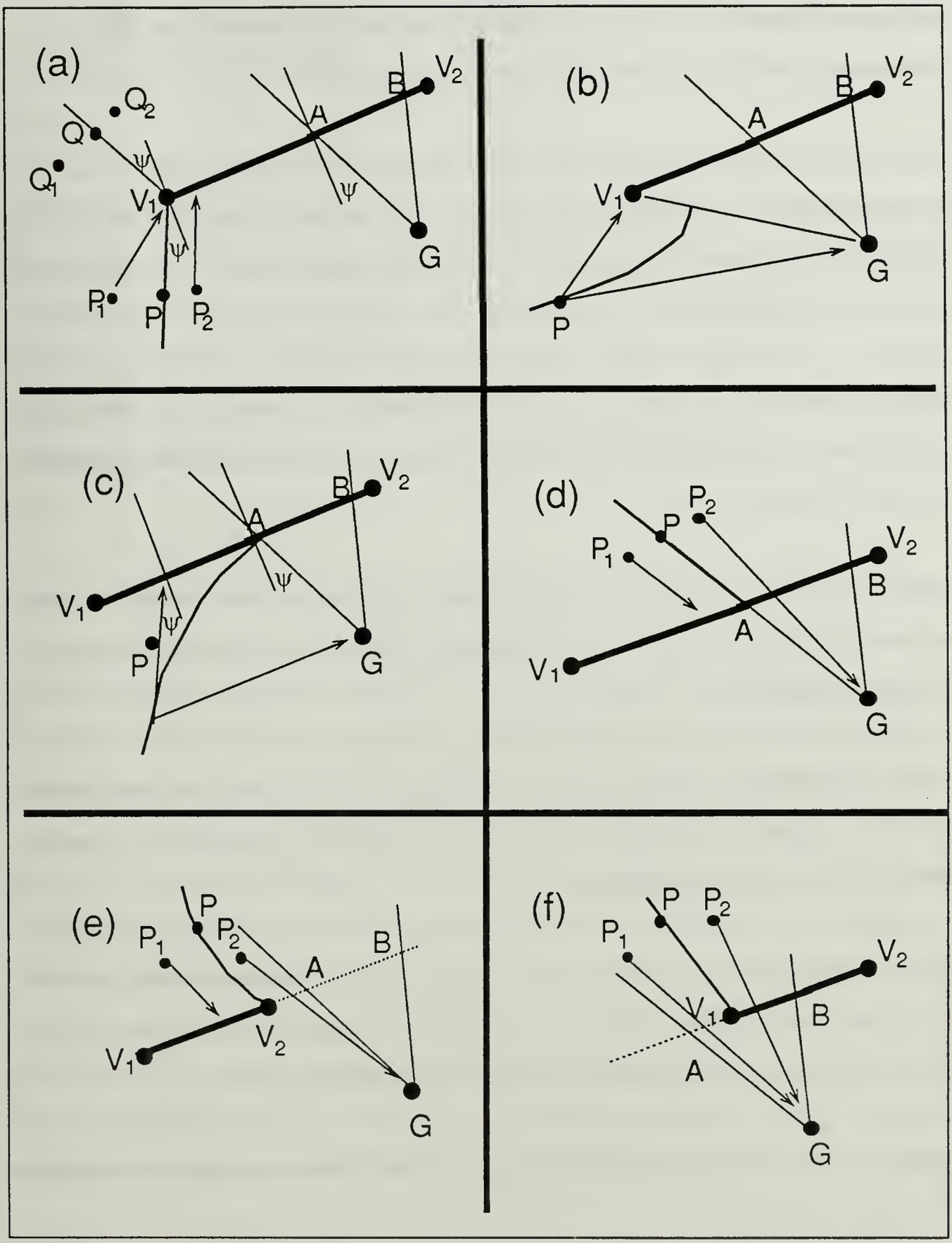


Figure 47
Road Segment Boundaries

in Equation 2. The boundary will begin at the point of intersection of the parabola with the road segment, which will be at V_2 if ∇AGB lies "outside" V_2, and will be at A otherwise.

PROOF V-3.4: From Figure 47c, the paths from P go to the road and travel along it, or go to the goal. Thus, the boundary is between regions with point root and road root, so Equation 2 applies. Since paths leave the road at W in Figure 37, point V_2 will correspond to point W if the wedge is "outside" V_2, or point A will correspond to point W otherwise. ♦

LEMMA V-3.5: Given road segment and goal G. If ∇AGB is "inside" road vertex V, a *road-travelling/road-crossing* boundary will be formed on the far side of the river which is a ray with vertex at point A and collinear with $\overline{GA}$ lying away from G.

PROOF: V-3.5: If ∇AGB is "inside" V, Figure 47d will apply. Paths from points P_1 just to the "left" of P in the figure will cross the road directly to G, while the path from P and P_2 enter the road and travel along it to A, where they exit to G. ♦

LEMMA V-3.6: Given a road segment with vertex V and goal G, with road cost-rate r_r and background cost-rate r_0. A *far-side-road-travelling/goal* boundary will exist if $\nabla$is outside V. The boundary will be a parabola which begins at V and lies away from the goal.

PROOF V-3.6: From Figure 47f, the paths from P go to the road and travel along it, or go to the goal. Thus, the boundary is between regions with point root and road root, so Equation 2 applies. Since the point W in Figure 37 is the point at which paths leave the road, V will correspond to point W. ♦

LEMMA V-3.7: If ∇AGB "straddles" V, a *road shadow* boundary will exist as a ray from V, collinear with $\overline{GA}$, and lying away from G.

PROOF V-3.7: From Figure 47g, a path from P_1 will cross the road, while a path from P_2 will bypass it. This will occur only if ∇AGB "straddles" V, because otherwise paths from P_2 will enter the road and travel along it to A. ♦

THEOREM V-3: Given a road segment V_1V_2 with cost-rate r_r, a goal G, and a background cost-rate r_0, if characteristic wedge ∇AGB is "inside" V_i, one road-end/road-travelling, two road-end, one near-side-road-travelling/goal, and one road- travelling/road-crossing boundaries exist on the V_i end of the road segment; when ∇AGB "straddles" V_i, a road shadow boundary exists on the V_i end; when ∇AGB is "outside" V_i, one near-side-road-travelling/goal and one far-side-road-travelling/goal boundaries exist on the V_i end; and the road segment is always a boundary. The form of these boundaries is as described in Lemmas V-3.1 through V-3.7.

PROOF V-3: Follows directly from Lemmas V-3.1 through V-3.7.♦

LEMMA V-4.1: Given high-cost, exterior-goal HCA with two visible edges V_1V_2 and V_3V_4, if the two regions whose paths cross the two edges are adjacent, the *visible-edge* boundary between them is described by Equation Set 4.

PROOF V-4.1: Per Figure 48a, the edges V_1V_2 and V_3V_4 are roots of region 1 and region 2 respectively. Paths which cross them go directly to G, and so the description of Theorem V-0.4 applies to this situation, and Equation Set 4 describes the boundary.♦

LEMMA V-4.2: Given high-cost, exterior-goal HCA with a visible edge V_1V_2 and a hidden edge V_3V_4, if the region whose paths cross edge V_1V_2 and the region whose paths go to and travel along edge V_3V_4 are adjacent, the *visible-hidden-edge* boundary between them is described by Equation Set 5.

PROOF V-4.2: Per Figure 48a, the edges V_1V_2 and V_5V_6 are roots of region 1 and region 3 respectively. Paths which cross edge V_1V_2 obey Snell's Law, and then go directly to G, while those which travel along edge V_5V_6

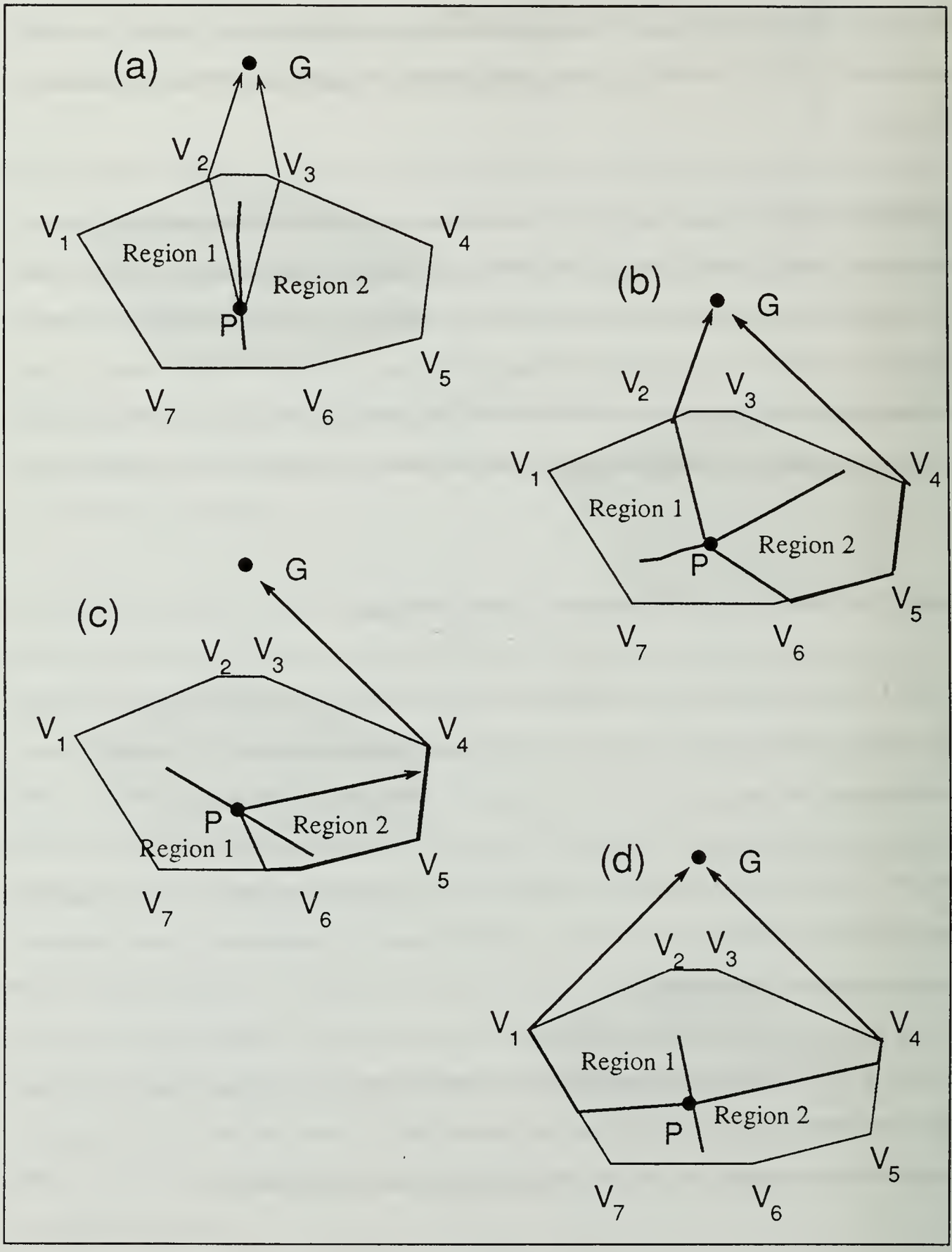


Figure 48
High-Cost Exterior-Goal HCA Boundaries

leave the edge at point V_5 en route to the goal and so the description of Theorem V-0.5 applies to this situation, and Equation Set 5 describes the boundary.♦

LEMMA V-4.3: Given high-cost, exterior-goal HCA with two hidden edges V_4V_5 and V_6V_7, such that $OPL(P_6) = [V_5, V_4 \mid OPL(P_4)]$, if the two regions whose paths enter and travel along the two edges are adjacent, the *hidden-edge-merging-path* boundary between them is described by Equation Set 3. ♦

PROOF V-4.3: Per Figure 48c, the edges V_4V_5 and V_6V_7 are roots of region 4 and region 5 respectively. Paths which enter edge V_4V_5 at the critical angle travel along it and leave at V_4 en route to the goal, while those which travel along edge V_6V_7 leave the edge at point V_6, eventually merging with paths from region 4. So the two edges are linearly- traversed edges and are the roots of regions 4 and 5, so the description of Theorem V-0.3 applies to this situation, and the boundary is a line segment as described therein.♦

LEMMA V-4.4: Given high-cost, exterior-goal HCA with two hidden edges V_4V_5 and V_6V_7, such that $OPL(P_6) = [V_5, V_4 \mid OPL(P_4)]$, if the two regions whose paths enter and travel along the two edges are adjacent, the *hidden-edge-diverging-path* boundary between them is a line segment described by Theorem V-0.3.

PROOF V-4.4: Per Figure 48e, the edges V_4V_5 and V_7V_8 are roots of region 4 and region 6 respectively. Paths which enter edge V_4V_5 at the critical angle travel along it and leave at V_4 en route to the goal, while those which travel along edge V_7V_8 leave the edge at point V_8 (going in the other direction around the exterior of the HCA). So the two edges are linearly- traversed edges and are the roots of regions 4 and 6, so the boundary between them is a line segment as described in Theorem V-0.3.♦

LEMMA V-4.5: Given a high-cost HCA with exterior goal G and vertices V_i. There is a boundary associated with each V_i such that optimal paths in one region cross edge $V_{i-1}V_i$ and optimal paths in the other region cross edge V_iV_{i+1}, except in the case that shorcutting occurs along the entire edge V_iV_{i+1} to edge $V_{i-1}V_i$, in which case no boundary occurs for vertex V_i.

PROOF V-4.6: There are three cases: both edges are visible, one edge is visible and the other is hidden, or both edges are hidden. When both edges are visible, by definition optimal paths from neither vertex includes points along an edge of the HCA. Consider Figure 30, and points near V_2 in the HCA interior. Since the interior has higher cost-rate than the exterior, there is no incentive for paths from points close to the visible edge to move further away from it. Rather, such paths will cross the edge as soon as possible to use the lower-rate exterior. Thus there are some points in the interior close to V_i whose paths cross edge $V_{i-1}V_i$ and some whose paths cross edge V_iV_{i+1}. There is, therefore, a boundary between them which begins at V_i and lies in the HCA interior.

In the second case, by the same reasoning as above, some paths whose start points are close to V_i will cross visible edge $V_{i-1}V_i$. But some points close to V_i may be far enough from edge $V_{i-1}V_i$ that it will be less costly to move initially away from the goal to edge V_iV_{i+1} in order to travel at the less expensive exterior rate. Clearly, this will cause a boundary which begins at V_i. If, however, edge V_iV_{i+1} forms an acute enough angle with $V_{i-1}V_i$ that there are no points near V_i for which it will be less costly to move away from the goal. In this situation, shortcutting will occur, at least in the vicinity of V_i. If some paths travel along edge V_iV_{i+1}, the point at which they shortcut into the interior will be the beginning of the boundary associated with V_i, because points just inside V_iV_{i+1} and toward V_{i+1} from the shortcutting point will have less costly paths by moving away from the goal to the lower-rate edge, while points just inside but toward V_i from the shortcutting point will go directly across the HCA. If shortcutting occurs all along edge V_iV_{i+1}, however, there will be no boundary associated with V_i, because all paths have the same behavior. In the third case, by the same reasoning as above, a vertex joining two hidden edges will have an associated boundary unless shortcutting occurs all along the edge. ♦

LEMMA V-4.7: The *edges* of a high-cost HCA with exterior goal are homogeneous-behavior boundaries.

PROOF V-4.7: Trivially true. ♦

LEMMA V-4.8: Given a high-cost HCA with exterior goal G, each vertex V of a hidden edge generates a linear *shadow boundary* which is the ray lying on the line defined by V and the first point P on OPL(V), starting at V and lying in the opposite direction from P.

PROOF V-4.8: The proof is the same as for Lemma V-1.2. (See Figures 23, 24, and 25.) ♦

LEMMA V-4.9: Given a high-cost HCA with exterior goal G with n interior boundary trees. There exists an *opposite-edge* boundary associated with each tree which begins at the point at which an interior boundary of the tree not associated with a vertex (i.e., not one of the leaf nodes of the tree, see Figures 30, 31, and 32) intersects an edge of the HCA. There is also an opposite-edge boundary which begins at each point at which two other opposite- edge, or a shadow and an opposite-edge boundary intersect. An opposite-edge boundary is described by Equation 1 if the interior boundary at which it begins separates regions of two linearly-traversed edges, or by Equation Set 6 if the interior boundary at which it begins separates regions whose paths cross two edges en route to the goal. If it begins at the intersection of two other exterior boundaries, it will be described by Equation 1 if the two regions which the intersecting boundaries do not have in common have point roots, and by Equation Set 6 (or a degenerate version) if one of the regions which the intersecting boundaries do not have in common has paths which cross two edges en route to the goal.

PROOF V-4.9: At the point at which an interior-boundary tree intersects a hidden edge of the HCA other than a vertex, one of four situations must exist. An optimal path from the point of intersection may go across the HCA interior and a second optimal path from the same point travels along the hidden edge, for example, in Figure 30 where two of the boundaries labelled "b" intersect edge V_4V_5. Secondly, one path from the point of intersection may cross a visible edge and a second path cross another edge, as in the boundary labelled "a" in Figure 30. Third, two paths may go from the point of intersection in opposite directions along the edge, as in the boundary labelled "d" in Figure 31, where one path goes through V_4 and one path goes through V_3. Fourthly, there may be only one optimal path from the point of intersection, as in the boundary in Figure 31 that intersects edge V_2V_3.

By examining Figures 30 and 31, it can be seen that when there are two optimal paths from the point of intersection of the interior boundary, there are points in the HCA exterior which also go in two directions, forming a boundary. In the fourth case above, where there is only one optimal path from the point of intersection, it can be seen that there is no exterior boundary. But the interior boundary in this case is associated with a verte x. In the first case, the exterior boundary separates a region whose points go to the vertex of the hidden edge through which goes the path from the intersection point, from the region whose paths cross two edges en route to the goal. This is a degenerate case of Theorem V-0.6, where one path crosses two edges and the other path goes through a point instead of crossing two edges, so Equation Set 6 applies. In the second case, the exterior boundary separates a region whose paths cross two edges from a region whose paths cross two other edges, so Equation Set 6 applies. In the third case, the exterior boundary separates two regions whose paths go through points, as in Theorem V-0.1 and Equation 1.

When any two exterior boundaries intersect, it must be that a third opposite-edge boundary begins, because past the point of intersection there must be a discrimination between the two regions which the first two boundaries did not have in common. The third boundary has as its region roots either two points, a point for one root and two edges for the other, or two different edges for both roots, because these are the only types of roots which the original exterior opposite-edge boundaries had. These roots are described by Equation 1 or Equation Set 6, where a degenerate case of Equation Set 6 is the case that one of the pair of edges is replaced by a vertex. Figures 30 and 31 show examples of exterior boundaries intersecting. ♦

LEMMA V-4.10: Given high-cost HCA with exterior goal G, and vertex V joining a visible and a hidden edge across which shortcutting occurs. There is a *corner-cutting* boundary which begins at point V and obeys the degenerate form of Equation Set 6 where paths on one side of the boundary cross two edges, while paths on the other side go through a vertex.

PROOF V-4.10: (See Figure 31.) Points on the shadow boundary emanating from V_2 in Figure 31 (labelled "e") go through V_2 to the goal. Since the HCA interior has a higher cost-rate than the exterior, there are some points just below the shadow boundary which will travel to V_2 rather than go through the HCA. But points

further below the shadow boundary will have further to go to vertex V_2 and so will cross the HCA, paying the higher cost-rate to do so. One set of points lies in a region with V_2 as root, while the other set of points lies in a region with edge V_2V_3 as root. Paths in the second region cross two edges en route to the goal. This conforms to the degenerate form of Equation Set 6. ♦

THEOREM V-4: A high-cost HCA with exterior goal has boundaries according to Lemmas V-4.1 through V-4.10.

PROOF V-4: Follows directly from Lemmas V-4.1 through V-4.10. ♦

LEMMA V-5.1: Given high-cost HCA with interior goal G. If the optimal path from a vertex V_i travels initially along an edge of the HCA, there is a *hidden-edge* boundary which begins at V_i and is a line segment conforming to Theorem V-0.3.

PROOF V-5.1: (See Figure 33.) Assume that for a vertex of high-cost interior-goal HCA V_i, $OPL(V_i) = [X, G]$, where X is a point on HCA edge V_iV_{i-1}, for example V_3 in Figure 33. Then there will be some points close to V_i in the HCA interior which will exit and travel along edge V_iV_{i-1} to X. Similarly, there will be some points close to V_i on edge $V_{i+1}V_i$ whose paths go through V_i, and so there will be points close to V_i in the HCA interior which exit the HCA and travel along edge $V_{i+1}V_i$ to V_i. Thus there are two regions in the vicinity of V_i, and the boundary between them separates paths which enter a linearly-traversed edge and travel along it from those which enter another linearly-traversed edge and travel along it. This is the situation of Theorem V-0.3, so the boundary is a line segment as described therein. ♦

LEMMA V-5.2: Given high-cost HCA with interior goal G. If the optimal path from a vertex V_i travels initially along an edge of the HCA, there is a *hidden-edge/goal* boundary which is a parabola as specified in Equation 2 which separates points which go to and travel along edge V_iV_{i+1} from points which go to and travel along edge V_iV_{i-1}.

PROOF V-5.2:(See Figure 33, boundaries labelled "b".) By the same reasoning as Lemma V-5.1, there are points close to vertex V_i which exit the region and travel along edge $V_{i+1}V_i$ to V_i. Consider point P which is moved away from V_i into the interior along the hidden-edge boundary associated with V_i. At some point, paths from P which go to edge V_iV_{i+1} will cost no less than a path from P straight to the goal at the higher cost rate. At this point, a new boundary begins separating points which go to edge V_iV_{i+1} and travel along it to V_i, from points which go to G. This is the same situation as described in Theorem V-0.2, with Equation 2 describing the parabolic boundary. ♦

LEMMA V-5.3: Given high-cost HCA with interior goal G. If the optimal path from a vertex V_i travels initially along edge V_iV_{i-1} of the HCA and cuts into the HCA at some point along edge V_iV_{i-1}, there is a *visible-edge/goal* boundary which is a parabola as specified in Equation 2 and separates points which travel along the visible edge V_iV_{i-1} from those which go directly to the goal.

PROOF V-5.3: By the same reasoning as Lemma V-5.2, when point P is far enough from V_i that paths which go to edge V_iV_{i-1} cost no less than a path that goes directly to G at the higher cost rate, a boundary will begin separating points which go to the linearly-traversed edge from those which go to the point G. This is the same situation as described in Theorem V-0.2, with parabola as described in Equation 2. ♦

LEMMA V-5.4: Given high-cost HCA with interior goal G, and two adjacent vertices V_i and V_{i+1} which have optimal paths lying on HCA edges, neither of which is edge V_iV_{i+1}. Then there will be an *interior-opposite-edge* boundary which is a line segment beginning on edge V_iV_{i+1} and conforming to the description of Theorem V-0.3.

PROOF V-5.4: If the optimal path from V_i lies initially on edge V_iV_{i-1}, and the optimal path from V_{i+1} lies initially on edge V_iV_{i+1}, as must be by assumption, there will be points in the interior of the HCA as described in Theorem V-0.3 which have paths which go to edge V_iV_{i+1} and travel along it to V_i, and similarly there will be points in the interior which have paths which go to edge V_iV_{i+1} and travel along it to V_{i+1}. Where these two regions meet, the boundary will separate points whose paths go to one linearly-traversed edge from points

whose paths go to another linearly- traversed edge, the situation described in Theorem V-0.3. Therefore, the boundary will be a line segment as described in Theorem V-0.3. ♦

LEMMA V-5.5: Each edge of a high-cost HCA with interior goal will be a *hca-edge* boundary.

PROOF V-5.5: Trivially true. ♦

LEMMA V-5.6: Given high-cost HCA with interior goal G. If the optimal path from a vertex V_i travels initially along an edge of the HCA with $OPL(V_i) = [X \mid OPL(X)]$, there is a *shadow* boundary which is a ray with vertex V_i and collinear with line V_iX, which lies away from X.

PROOF V-5.6: The proof proceeds as in Proof V-1.2.♦

LEMMA V-5.7: Given a high-cost HCA with interior goal G, and opposite edge V_iV_{i+1} as defined in Lemma V-5.4. Then an exterior *opposite-edge* boundary exists which conforms to Equation 1.

PROOF V-5.7: At the point at which the interior-opposite-edge boundary intersects edge V_iV_{i+1}, there are two optimal paths which go through vertices V_i and V_{i+1}. Points will exist in the exterior, but close to this intersection point, which will have optimal paths which go through these vertices as well. These points are on a boundary which separates points whose paths go through V_i from those which go through V_{i+1}, two regions with point roots. Therefore, Theorem V-0.1 applies, and the boundary is a hyperbola segment which conforms to Equation 1.♦

LEMMA V-5.8: Given high-cost HCA with interior goal G, and vertex V_i which has optimal path which goes directly to G. There will be a *visible-edge* boundary in the HCA exterior beginning at V_i which conforms to Equation Set 4.

PROOF V-5.8: Consider points close to V_i outside the HCA. Since the best path from V_i is straight to the goal, clearly paths from points in the lower-cost exterior will have optimal paths which go directly to the goal via

a Snell's-Law path across one of the edges incident upon V_i. The boundary which separates paths which cross one edge from those which cross the other edge conform to the situation described in Theorem V-0.4, and so the boundary will conform to Equation Set 4. ♦

LEMMA V-5.9: Given a high-cost HCA with interior goal G and vertex V_i with associated hidden-edge/goal boundary which intersects edge V_iV_{i+1}. Then there will be a *corner-cutting* boundary which begins at the point of intersection and continues into the exterior conforming to a degenerate form of Equation Set 4, where one edge-crossing degenerates to a point crossing.

PROOF V-5.9: (See Figure 33, boundaries labelled "h".) At the point of intersection of the hidden-edge/goal boundary with edge V_iV_{i+1}, there are two optimal paths; one goes directly to the goal, and the other goes through V_i. A point just outside the HCA in the vicinity of the point of intersection may therefore have a path which goes to V_i, or which crosses edge V_iV_{i+1} en route to the goal. The boundary separating such points is therefore a boundary between a region which has a point as root, and one which has an edge-crossing as root. This is a degenerate form of the situation of Theorem V-0.4, so Equation Set 4 applies. ♦

THEOREM V-5: Given a high-cost HCA with interior goal, the boundaries associated with the HCA are as described by Lemmas V-5.1 through V-5.9.

PROOF V-5: (See Figure 33.) Follows directly from Lemmas V-5.1 through V-5.9. ♦

LEMMA V-6.1: Given a low-cost HCA with interior goal point G, there are no boundaries in the HCA interior.

PROOF V-6.1: (See Figure 34.) Assume that there is a point P with optimal path OPL(P) = [R | OPL(R)], i.e., that the path does not go directly to the goal. R must lie on an edge or vertex, by Theorem I-2. In either case, the path must be longer in Euclidean distance than the line segment $\overline{PG}$, by the triangle inequality. Since the interior cost-rate is lower than the exterior cost-rate, there is no advantage to a path to use the exterior cost-

rate, so the cost of $\overline{PG}$ must be less than $|\overline{PR}| + |(RG)^*|$, which is a contradiction. Therefore all interior points have the path list [G]. By the definition of a homogeneous-behavior region, the entire HCA interior is a single region, so there are no interior boundaries. ♦

LEMMA V-6.2: Given a low-cost HCA with interior goal G. From each vertex V there are two *vertex/edge-crossing* boundaries separating points whose optimal paths go through V and then to G from those which cross an edge obeying Snell's-Law and then go to G. Each boundary lies on the the path from G through V which obeys Snell's Law for crossing one of the edges incident upon V.

PROOF V-6.2:(See Figure 34.) Consider a point P in the HCA exterior arbitrarily close to the exterior leg of a Snell's-Law path from G through V with respect to edge E. The optimal path from P goes through edge E obeying Snell's Law. By the principle of optimality (Theorem I-1), all points along that path also have optimal paths which lie on the same path. Thus, the boundary is a ray lying at the angle prescribed by Snell's Law. ♦

THEOREM V-6: Given a low-cost HCA with interior goal. The interior has no boundaries, and the exterior boundaries are as described in Lemma V-6.1.

PROOF V-6: Follows directly from Lemmas V-6.1 and V-6.2.♦

LEMMA V-7.1: Given a low-cost HCQ with exterior goal, each edge is an *hca-edge* boundary.

PROOF V-7.1: Trivially true.♦

LEMMA V-7.2: Given low-cost HCA with exterior goal G and vertex V such that the optimal path from V goes initially into the HCA interior. Then a *vertex/edge-crossing* boundary exists for each edge incident upon V which is the second leg of a path from G through V which obeys Snell's Law with respect to the edge, and separates paths starting in the exterior which go through V from paths which cross the edge. If the optimal path from V goes initially along an edge of the HCA, one such boundary exists with respect to the edge incident upon V not travelled by the path from V.

PROOF V-7.2: (See Figure 35.) The same reasoning as in Proof V-6.2 applies here. ♦

LEMMA V-7.3: Given low-cost HCQ with exterior goal G, interior cost-rate r_i, exterior cost-rate r_e, and vertex V such that the optimal path from V goes initially along an edge of the HCA incident upon V. Then a *vertex/edge-following* boundary exists which is a ray from V along a line which makes the angle $\pi/2 + \theta_c$ with the edge, where $\theta_c = \sin^{-1}(r_i/r_e)$.

PROOF V-7.3: (See Figure 35.) The analysis is the same as Proof V-6.2 above. ♦

LEMMA V-7.4: Given low-cost HCA with exterior goal G, and vertex V with optimal path which goes initially along an edge of the HCA. There is a parabolic *edge-following/goal* boundary which begins along the edge, conforms to Equation 2, and separates paths which go to the edge and follow it, from paths which go directly to the goal.

PROOF V-7.4: (See Figure 35.) The proof is the same as for the near-side-road- travelling/goal boundary for road segments in Proof V-3.4. ♦

LEMMA V-7.5: Given low-cost HCA with exterior goal G, and vertex V such that the optimal path from V lies along an edge of the HCA incident upon V. Then there is a hyperbolic *vertex/goal* boundary which conforms to Equation 1, and separates paths which go through V from those which go directly to G.

PROOF V-7.5: (See Figure 35.) The proof is the same as for road-end/goal boundary of road segments, Proof V-3.3. ♦

LEMMA V-7.6: Given a low-cost HCA with exterior goal and vertex V_i such that the optimal path from V_i lies in the HCA interior, and vertex V_{i-1} adjacent to V_i and closer to G. A *edge-crossing/goal* will exist if the vertex/goal boundary associated V_{i-1} intersects the both vertex/edge-following boundaries emanating from V_{i-1}. It will conform to a degenerate form of Equation Set 6, and separate paths which cross edge V_iV_{i-1} and then cross a visible edge en route to the goal, from paths which go straight to the goal.

PROOF V-7.6: (See Figure 35.) At the point at which the hyperbolic vertex/goal boundary intersects the vertex/edge-following boundary associated with edge V_iV_{i-1}, the two regions not common to the boundaries are the one whose paths go straight to the goal, and the one whose paths cross the edge en route to a second edge crossing, and the goal. But this is the form of Theorem V-0.6, where one pair of edge-crossings degenerates to a single point- crossing. Thus Equation Set 6 applies. ♦

LEMMA V-7.7: Given low-cost HCA with exterior goal G, and vertex V with optimal path which goes directly to the goal, such that V is not incident to any other homogeneous-behavior-region boundaries. There is a *visible- edge* boundary in the HCA interior which begins at V and continues across the HCA to a hidden edge.

PROOF V-7.7: Consider points inside the HCA near V. The path from such a point crosses one edge incident upon V or the other (See Figure 35). Therefore, there are two regions inside the HCA, and the boundary separates the two. Since the region roots are both edges crossed by paths, Theorem V-0.4 applies, so the boundary conforms to Equation Set 4. ♦

LEMMA V-7.8: Given low-cost HCA with exterior goal G, and vertex V with optimal path which goes directly to the goal, such that V is not incident to any other homogeneous-behavior-region boundaries, and given the *visible- edge* boundary in the HCA interior as specified in Lemma V-7.7. There is an *opposite-edge* boundary in the HCA exterior which begins at the point of intersection of the *visible-edge* boundary with the hidden edge and conforms to Equation Set 6.

PROOF V-7.8: At the point of intersection of the visible-edge boundary with the hidden edge, there are two optimal paths, which cross the two edges incident upon V. Points in the HCA exterior near this point of intersection will cross into the HCA interior, crossing on one side or the other of the point of intersection. Points which cross on one side will traverse the HCA interior and cross one of the edges incident upon V, while points which cross on the other side will cross the other edge incident upon V. Therefore, the boundary which separates points with these two behaviors conforms to Equation Set 6. ♦

THEOREM V-7: Given a low-cost HCA with exterior goal, boundaries are generated according to Lemmas V-7.1 through V-7.8.

PROOF V-7: Follows directly from Lemmas V-7.1 through V-7.8. ♦

APPENDIX B - POINT-TO-POINT WAVEFRONT PROPAGATION ALGORITHM

```
algorithm wavefront-propagation                    (Algorithm B-1)
   input: Start-Point, Goal-Point
   {
   Wavefront := Start-Point;
   while (Status = INPROGRESS)                     /* iteratively expand wavefront until */
      expand-wavefront(Wavefront);                 /* status is DONE or NIL */
   if (Status = DONE)
      Optimal-Path := Goal-Point concatenated
         with back-path(Goal-Point);
   else                                            /* status is NIL, so no feasible solution */
      Optimal-Path is undefined;
   }                                               /* end of wavefront-propagation */

procedure expand-wavefront
   input: Wavefront
   {
   if (Wavefront is empty)                         /* Base case of recursion. If empty at 1st call */
      Status := NIL;                               /* to expand-wavefront, there is no feasible path */
   else
      {
      Current-Cell := cell on Wavefront with min remaining cost;
      expand-cell(Current-Cell);
      if not (Status = DONE)
         {
         Rest-of-Wavefront := Wavefront less Current-Cell;
         expand-wavefront(Rest-of-Wavefront);     /* recursive call to expand-wavefront */
         if not (Status = DONE)
            {
            Wavefront := Cells-for-New-Wavefront  /* Note: Wavefront is recursively emptied */
               appended onto front of Wavefront;  /* out level by level and new Wavefront */
            Status := INPROGRESS;                 /* is built up as each level returns. */
            }
         }
      }
   }                                               /* end of expand-wavefront */
```

```
procedure expand-cell
   input: Current Cell
   {                                                  /* initialize flag to assume that Current-Cell */
   Finished-With-Cell := TRUE;                        /* will not stay on Wavefront */
   Cells-for-New-Wavefront := empty list;
   for (New-Cell := North-, East-, South-, and West-Neighbor)
      orthogonal-expand(Current-Cell,New-Cell);
   for (New-Cell := Northeast-, Southeast-, Southwest-, and Northwest-Neighbor)
      diagonal-expand(Current-Cell,New-Cell);
   if not (Finished-With-Cell)                        /* keep Current-Cell on Wavefront */
      Cells-for-New-Wavefront := Current-Cell appended
         onto Cells-for-New-Wavefront;
   if (Cells-for-New-Wavefront contains Goal-Point)
      Status := DONE;
   else
      Status := INPROGRESS;
   }                                                  /* end of expand-cell */

procedure orthogonal-expand
   input: Current-Cell, New-Cell
   {
   if ((Parent-Pointer-of-New-Cell is not yet set)    /* if this is first cell to expand into New- */
      or (Parent-Pointer-of-New-Cell = Current-Cell)  /* Cell,  or this path costs less to expand into */
*/    or ((Initial-Cost-of-New-Cell - 1.414)          /* New-Cell, set backpointer and explore New-Cell.
         < Cost-of-New-Cell))
      {
      Parent-Pointer-of-New-Cell := Current-Cell;     /* Current-Cell becomes parent of New-Cell. */
      Cost-of-New-Cell := Cost-of-New-Cell - 1.414;   /* decrement cost of New-Cell */
      if (Cost-of-New-Cell < 0)
         {                                            /* if New-Cell has been fully explored, */
         overflow(Current-Cell, New-Cell);            /* then New-Cell and possibly an overflow */
         Cells-for-New-Wavefront := Overflow-List     /* cell are added to new Wavefront */
            appended onto New-Cell;
         }
      else
         {                                            /* if New-Cell has not been fully explored, */
         Cells-for-New-Wavefront := empty list;       /* New-Cell is not added to new Wavefront */
         Finished-With-Cell := FALSE;                 /* but reset the flag to note that */
         }                                            /* Current-Cell must stay on Wavefront */
      }
   }                                                  /* end of orthogonal-expand */
```

```
procedure diagonal-expand
   input: Current-Cell, New-Cell
   {
   if ((Parent-Pointer-of-New-Cell is not yet set)          /* if this is first cell to expand into New- */
      or (Parent-Pointer-of-New-Cell = Current-Cell)        /* Cell, or this path costs less to expand into */
*/    or ((Initial-Cost-of-New-Cell - 1.0)                  /* New-Cell, set backpointer and explore New-Cell.
         < Cost-of-New-Cell))
      {
      Parent-Pointer-of-New-Cell := Current-Cell;           /* Current-Cell becomes parent of New-Cell. */
      Cost-of-New-Cell := Cost-of-New-Cell - 1.0;           /* decrement cost of New-Cell. */
      if (Cost-of-New-Cell < 0)                             /* if New-Cell is fully explored, */
         Cells-for-New-Wavefront := Cells-for-New-          /* add it to new Wavefront. */
            Wavefront appended onto New-Cell;
      else
         {                                                  /* if New-Cell is not fully explored,*/
         Cells-for-New-Wavefront := null list;              /* do not add it to new Wavefront */
         Finished-With-Cell := FALSE;                       /* and reset flag to insure that Current-Cell */
         }                                                  /* gets put back on Wavefront. */
      }
   }                                                        /* end of diagonal-expand */

procedure overflow
   input: Current-Cell, New-Cell
   {
   Overflow-Cell := cell on opposite side of New-Cell from Current-Cell;
   if ((Parent-Pointer of Overflow-Cell is not yet set)
      or (Parent-Pointer of Overflow-Cell = New-Cell)
      or ((Initial-Cost-of-New-Cell - 1.0)
         < Cost-of-New-Cell))
      {
      Parent-Pointer of Overflow-Cell := New-Cell;          /* Current-Cell becomes parent of New-Cell. */
      Cost of Overflow-Cell := Cost of                      /* decrement Overflow-Cell by the negative */
         Overflow-Cell + (Cost of New-Cell);                /* amount left over from New-Cell. */
      if (Cost of Overflow-Cell < 0)                        /* if necessary, call  overflow again. */
         {
         overflow(New-Cell, Overflow-Cell);
         Overflow-List := Overflow-List
            appended onto Overflow-Cell;
         }
      else                                                  /* else Overflow-Cell is not */
         Overflow-List := empty list;                       /* added to new Wavefront. */
      }
   else                                                     /* if Overflow-Cell already has */
      Overflow-List := empty list;                          /* a parent, do nothing. */
   }                                                        /* end of overflow */
```

APPENDIX C - WAVEFRONT-PROPAGATION OPM CONSTRUCTION SOURCE CODE

```
;*************************************************************************
;*************************************************************************
;***** "opm" creates an optimal path map by finding the boundaries
;***** between regions of similarly-behaved optimal paths using the
;***** wavefront propogation algorithm.  The basic structure of the
;***** wavefront algorithm used is adapted from a Prolog program
;***** by MAJ Bob Richbourg, June 87.
;***** This is the "pure" version which tests for boundaries by checking
;***** for the equivalent turn-points in the optimal-path list of
;***** neighboring cells.
;*****
;***** Current as of 27 Jun 89
;*****
;***** Input: files "declar", "initmap", "utils", "bdry", & "graphics".
;***** Output: Graphical output to the host Symbolics screen.
;*************************************************************************
;*************************************************************************

;*************************************************************************
;***** Function "opm" is the top-level function of file opm.lisp
;*****   Arguments: none
;*****   Returned: T.
;*****   Side Effects: sets *boundary* array with the pixels which
;*****     represent region boundaries.
;*****   Functions Used: initialize-map, initialize-graphics,
;*****     expand-wavefront, draw-and-show-windows,
;*****     draw-and-show-bdry-window, kill-windows,
;*****     and report-completion.
(defun opm ()
  (setf *internal-time1* (get-internal-run-time))
  (setf *external-time1* (get-universal-time))
  (initialize)
  (princ "Init Process Time: ")
  (prin1 (- (setf *internal-time2* (get-internal-run-time)) *internal-time1*))
  (linefeed)
  (princ "     Elapsed Time: ")
  (prin1 (- (setf *external-time2* (get-universal-time)) *external-time1*))
  (linefeed)
  (do ((Wavefront (list *goal*)
       (expand-wavefront Wavefront)))
      ((null Wavefront)))
;     (draw-and-show-window))
  (cond ((equal nil *incremental-bdry-check*) (check-all-boundaries)))
  (princ " Expansion Process Time: ")
  (prin1 (- (setf *internal-time1* (get-internal-run-time)) *internal-time2*))
  (linefeed)
  (princ "           Elapsed Time: ")
  (prin1 (- (setf *external-time1* (get-universal-time)) *external-time2*))
  (linefeed)
  (draw-and-show-bdry-window)
  (cond ((null *incremental-bdry-check*) (show-backpaths)))
  (report-completion))

;*************************************************************************
;***** Function initialize loads files, preprocesses the map, and
;*****   initializes the graphics screen.
(defun initialize ()
  (load "declar")
  (load "initmap")
  (load "utils")
  (load "bdry")
  (load "graphics")
```

```
  (initialize-map)
  (initialize-graphics)
  (princ "Beginning Wavefront Expansion")(linefeed)(linefeed))

;*************************************************************************
;***** Function expand-wavefront: computes the next wavefront by taking
;*****              the first pair of cell coordinates from the wave
;*****              and processing it, then recursively processing
;*****              the rest of the list in the same manner.
;*****    Argument: Wave, the remainder of the old wavefront left to process
;*****    Returned: the new wavefront, or nil if Wave becomes empty
;*****    Side Effects: see below
;*****    Functions Used: expand-cell and expand-wavefront
(defun expand-wavefront (Wave)
  (cond ((null Wave) nil)
        (t (append (expand-cell (car Wave))
                   (expand-wavefront (cdr Wave))))))

;*************************************************************************
;***** Function expand-cell: determines which of the eight neighboring
;*****       cells will be on the new wavefront and whether there is a
;*****       region boundary around the center cell.
;*****    Argument: Cell, a list of the X,Y coords of the cell on
;*****       the current wavefront being processed.
;*****    Returned: A list of cells to be added to the new wavefront
;*****    Side Effects: none
;*****    Functions Used: orthog-expand, diag-expand
;*****
(defun expand-cell (Cell)
  (setq *finished-with-cell-p* 't)    ; initialize flag - assume
                                      ; cell will not stay on wf
  (cond ((not (null *incremental-bdry-check*)) (check-for-boundaries Cell)))
  (let* ((X (car Cell))
         (Y (cadr Cell))
         (Cells-to-add
           (nreverse
             (remove nil
               (append
                 (orthog-expand (list X (1+ Y)) (list X Y))
                 (orthog-expand (list (1+ X) Y) (list X Y))
                 (orthog-expand (list X (1- Y)) (list X Y))
                 (orthog-expand (list (1- X) Y) (list X Y))
                 (diag-expand (list (1- X) (1+ Y)) (list X Y))
                 (diag-expand (list (1+ X) (1+ Y)) (list X Y))
                 (diag-expand (list (1+ X) (1- Y)) (list X Y))
                 (diag-expand (list (1- X) (1- Y)) (list X Y)))))))
        (cond
           ((null *finished-with-cell-p*)     ; If some neighbors are not fully
            (cons (list X Y) Cells-to-add))  ; explored, leave center cell on wf
           (t Cells-to-add))))

;*************************************************************************
;***** Function diag-expand: explores a cell which is in a diagonal
;*****       direction from the cell being expanded.
;*****    Arguments: same as orthog-expand
;*****    Returned: A list consisting of a list of cells to be added to
;*****       the new wavefront and a flag to note that (1) Center-cell
;*****       has fully explored its neighbor, or (0) it has not.
;*****    Side Effects: Sets the parent coords of New-cell if they are nil
(defun diag-expand (New-cell Center-cell)
  (let ((Xn (car New-cell))
        (Yn (cadr New-cell))
        (Xc (car Center-cell))
        (Yc (cadr Center-cell)))
```

```
    (cond ((null (aref *cell* Xn Yn 1))          ; If New-cell not explored
           (setf (aref *cell* Xn Yn 1)           ; yet, and is not an obstacle,
                 Center-cell)                    ; Center-cell becomes its parent
           (set-opl Xn Yn Xc Yc)                 ; Set Opt-Path-List for (Xn,Yn)
           (setf (aref *cell* Xn Yn 0)               ; Decrement cost
                 (- (aref *cell* Xn Yn 0) 1))
           (cond ((<= (aref *cell* Xn Yn 0) 0)     ; If Newcell is fully explored
                  (setq *backpath-pixel-list*      ;
                    (append                        ;
                      (get-backpath Xn Yn)         ; add its parent to the
                      *backpath-pixel-list*))      ; display list of parents
                  (list New-cell))                 ; and add New-cell to wave.
                 (t (setq *finished-with-cell-p*     ; If New-cell is not fully
                          nil))))                    ; explored, don't add to wf,
                                                     ; and note that Center-cell
                                                     ; must stay on wavefront.
          ((and (= Xc (car (aref *cell* Xn Yn 1)))      ; If Newcell's parent is
                (= Yc (cadr (aref *cell* Xn Yn 1)))     ; Center-cell and Newcell
                (> (aref *cell* Xn Yn 0) 0))            ; not fully explored,
             (setf (aref *cell* Xn Yn 0)                        ; Decrement cost.
                   (- (aref *cell* Xn Yn 0) 1))
             (cond ((<= (aref *cell* Xn Yn 0) 0)  ; If Newcell is fully explored
                    (setq *backpath-pixel-list*
                      (append                    ; Add parents to the
                        (get-backpath Xn Yn)     ; backpath display
                        *backpath-pixel-list*))
                    (list New-cell))             ; Add current new cell to wf
                   (t (setq *finished-with-cell-p*   ; If New-cell is not fully
                            nil))))                  ; explored don't add it to
                                                     ; wf,and note that Center
                                                     ; must stay on wavefront.
          (t nil))))       ; If Newcell was already explored, don't add to wave.

;*****************************************************************************
;***** Function orthog-expand: explores a cell which is in an orthogonal
;*****      direction from the cell being expanded.
;*****   Arguments: the first argument is a list of the X,Y
;*****      coords of the cell being explored; the second is a list
;*****      of coordinates of the cell on the current wavefront
;*****      being expanded from.
;*****   Returned: A list of two elements: the first is a list of
;*****      new cells to be added to the new wavefront and the second
;*****      is a flag set as indicated above (in diag-expand)
;*****   Side Effects: Sets the parent coords of New-cell if they are nil
(defun orthog-expand (New-cell Center-cell)
  (let ((Xn (car New-cell))
        (Yn (cadr New-cell))
        (Xc (car Center-cell))
        (Yc (cadr Center-cell)))
    (cond ((null (aref *cell* Xn Yn 1))          ; If New-cell not explored
           (setf (aref *cell* Xn Yn 1)           ; yet, and is not an obstacle,
                 Center-cell)                    ; Center-cell becomes its parent
           (set-opl Xn Yn Xc Yc)                ; Set Opt-Path-List for (Xn,Yn)
           (setf (aref *cell* Xn Yn 0)                ; Decrement cost
                 (- (aref *cell* Xn Yn 0) 1.414))
           (cond ((<= (aref *cell* Xn Yn 0) 0)     ; If New-cell is fully explored
                  (setq *backpath-pixel-list*      ;
                    (append                        ; Add its parent to the
                      (get-backpath Xn Yn)         ; display list
                      *backpath-pixel-list*))      ;
                  (append (overflow                ; Explore next cell in dir-
                            New-cell               ; ection of expansion & add
                            Center-cell)           ; any overflow cells.
                          (list New-cell)))        ; Add current new cell to wf
```

```
                                          ; to right of overflow cells.
            (t (setq *finished-with-cell-p*      ; Else if Newcell not
                     nil) nil)))                 ; fully explored don't add
                                                 ; it to wf,and note that
                                                 ; Center-cell stays on wf.
      ((and (= Xc (car (aref *cell* Xn Yn 1)))      ; If Newcell's parent is
            (= Yc (cadr (aref *cell* Xn Yn 1)))     ; Center-cell:
            (> (aref *cell* Xn Yn 0) 0))
         (setf (aref *cell* Xn Yn 0)                        ; Decrement cost.
               (- (aref *cell* Xn Yn 0) 1.414))
         (cond ((<= (aref *cell* Xn Yn 0) 0)    ; If Newcell is fully explored
                (setq *backpath-pixel-list*
                  (append                       ; Add parents to the
                    (get-backpath Xn Yn)        ; backpath display
                    *backpath-pixel-list*))
                (append (overflow             ; Explore next cell in dir-
                          New-cell            ; ection of expansion & add
                          Center-cell)        ; any overflow cells.
                        (list New-cell)))     ; Add current new cell to wf
                                              ; to right of overflow cells.
               (t (setq *finished-with-cell-p*    ; If New-cell is not fully
                        nil) nil)))               ; explored don't add it to
                                                  ; wf,and note that Center
                                                  ; must stay on wavefront.
      (t nil))))       ; If Newcell was already explored, don't add it to wf.


;*************************************************************************
;***** Function overflow: determines whether expansion should continue
;*****   into the next cell in the (orthogonal) direction in which
;*****   it has been going, and expands if necessary.
;*****   Arguments: the first is a list of the X,Y coords of the cell
;*****     into which the wave will overflow; the second is the coords
;*****     of the cell from which it overflowed.
;*****     Note that Center-cell in this function is the variable
;*****     called New-cell in orthog-expand, and Parent-cell here is
;*****     called Center-cell in orthog-expand.
;*****   Returned: A list of cells to add to wavefront
;*****   Side Effects: cell costs are decremented
(defun overflow (Center-cell Parent-cell)
  (let* ((Xc (car Center-cell))
         (Yc (cadr Center-cell))
         (Xp (car Parent-cell))
         (Yp (cadr Parent-cell))
         (Xn (+ Xc (- Xc Xp)))          ; Explore the next cell in the direction
         (Yn (+ Yc (- Yc Yp)))          ; of the previous expansion
         (New-cell (list Xn Yn))
         (overflow-cost                                 ; Check if overflow is at
                                                        ; a boundary;
           (cond ((null (aref *cell* Xn Yn 0)) 0)       ; if not,decrement overflow
                                                        ; cell by the (negative)
                 (t (+ (aref *cell* Xc Yc 0)            ; amount left over from
                       (aref *cell* Xn Yn 0))))))       ; previous cell.
    (cond ((null (aref *cell* Xn Yn 1))     ; If overflow cell is unexplored,
           (setf (aref *cell* Xn Yn 1)      ; Set overflow cell parent
                 Center-cell)               ; to the explored cell.
           (set-opl Xn Yn Xc Yc)            ; Set Opt-Path-List for (Xn,Yn)
           (setf (aref *cell* Xn Yn 0)
                 overflow-cost)
           (cond ((< overflow-cost 0)
                  (setq *backpath-pixel-list*
                    (append                         ; Add parent to the
                      (get-backpath Xn Yn)          ; backpath display.
                      *backpath-pixel-list*))
```

```
        (append (overflow          ; If more overflow, expand again,
                  New-cell        ; and add Newcell to wave list.
                  Center-cell)
                (list New-cell)))
       (t nil)))                    ; Else put nothing on wavefront.
(t nil))))                         ; Else put nothing on wavefront.
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER -*-


;*********************************************************************
;*********************************************************************
;***** "declar" contains the declarations of global variables used
;***** by "opm".  It is loaded by function "opm" in file "opm.lisp".
;***** This version is for use with maps in the form of a rectangle
;***** of characters.
;*****
;***** Current as of 7 Jun 89
;*****
;***** Side Effects: initialization of *cell* and *boundary* arrays,
;*****                and other global variables as listed below.
;*********************************************************************
;*********************************************************************



;***** Global Variables:

(defvar *version*)
;(setf *version* "pure")(setf *vertex-list* nil)(setf *edge-list* nil)
(setf *version* "vertex-edge")
;(setf *version* "diverging-path")(setf *vertex-list* nil)(setf *edge-list* nil)
(defvar *incremental-bdry-check*)     ;***** set to 't if check-boundaries should
(setq *incremental-bdry-check* 't)    ;***** be done at each expand-cell, nil if not.
(defvar *internal-time1*)
(defvar *external-time1*)
(defvar *internal-time2*)
(defvar *external-time2*)
(defvar *map-width*)      ;***** Max allowable number of columns in the
(setq *map-width* 205)    ;       map + 2 for bordering columns of blanks
(defvar *map-length*)     ;***** Max allowable number of lines in the
(setq *map-length* 155)   ;       map (=153) + 2 (=155) for the bordering lines of blanks
(defvar *magnification*) ;***** Magnification of the screen.
(setq *magnification* 3) ;
(defvar *river-cost*)     ;***** Cost to cross a river
(setq *river-cost* 16)
(defvar *road-cost*)      ;***** Cost to use a road
(setq *road-cost* 0.1)
(defvar *mapline*)              ;***** Array to hold the input map: each element
(setf *mapline*                 ;      is a string, each of whose characters
      (make-array               ;      represents one cell of the map.
       (list *map-length*)))
(defvar *terrain-pixel-list*)       ; List to hold coordinates
(setq *terrain-pixel-list* nil)     ;      of terrain pixels.
(defvar *boundary-pixel-list*)      ; List to hold coordinates
(setq *boundary-pixel-list* nil)    ;      of boundary pixels.
(defvar *backpath-pixel-list*)      ; List to hold coordinates
(setq *backpath-pixel-list* nil)    ;      of backpath pixels.
(defvar *finished-with-cell-p*)     ; Flag to record if cell stays on wave.
(defvar *output-stream*)            ; Can be used to define output stream
(defvar *goal*)           ;***** coordinates of goal point
(defvar *cell*)           ;***** 3-dimen array whose first and second indices
(setf *cell*              ;      are the cell coordinates and whose third index
      (make-array         ;      specifies the attribute:
        (list             ;      Attribute 0 is cost to traverse the cell,
           *map-width*    ;      decremented as wave passes over cell,
           *map-length*   ;      Attribute 1 is list of parent's coords
           4)))           ;      if specified, nil if not.
                          ;      Attribute 2 is list consisting of the
                          ;      character symbol of the cell, followed if
                          ;      if applicable by an edge id and vertex flag
```

```
                          ;       Attribute 3 is coords of opt-path-list parent
(defvar *boundary*)       ;***** Bit-valued array to mark region boundaries.
(setf *boundary*          ;       The (X,Y,0) element specifies whether there is
      (make-array         ;       a boundary to immediate right of cell (X,Y).
        (list             ;       The (X,Y,1) element specifies whether there is
           *map-width*    ;       a boundary immediately below cell (X,Y).
           *map-length*   ;       Altho this array has enough info to specify
           2)             ;       boundaries between pixels, pixel (X,Y) is
      :element-type 'bit)) ;      plotted as the boundary.
(defvar *edge-list*)      ;***** These lists are for the heuristic version, and
(defvar *vertex-list*)    ;       list all edge cells with edge id & vertex cells.
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER -*-

;*********************************************************************
;*********************************************************************
;***** "initmap" contains the functions used by "opm" to examine
;***** the map symbols and encode them into elements of the
;***** *cell* array.  It is loaded by function "opm" in file "opm".
;***** This version is for use with maps in the form of a rectangle
;***** of characters.
;*****
;***** Current as of 8 Jun 89
;*****
;***** Input: file "map", an array of cell attributes
;*****
;***** Side Effects: Loads file "map.lisp", and sets the
;*****         elements of the *cell* array according to the
;*****         associated map symbol. Adjusts *map-width*.
;*********************************************************************


;*********************************************************************
;***** Function "initialize-map" initializes the values of the
;*****      array *cell* according to the information encoded
;*****      in graphic form in the file "map".
;*****    Arguments: none
;*****    Returned: t in all cases.
;*****    Side Effects: Loads the Lisp file "map".
;*****                  Initializes the arrays *cell* and *boundary*
;*****                  and adjusts the variable *map-width*.
;*****    Functions Used: process-line, process-char
(defun initialize-map ()
    (load "map")(linefeed)(linefeed)
    (princ "Initializing Map")(linefeed)
    (cond ((equalp *version* "vertex-edge")
           (process-vertex-info *vertex-list*)
           (process-edge-info *edge-list*)))
    (setq *map-width* (+ 2 (length (aref *mapline* 1))))
    (do ((I 0 (1+ I)))
           ((= *map-width* I))                ; Initialize the top
           (process-char #\x I 0))            ; "buffer zone" row
    (do((J 1 (1+ J)))
       ((string-equal "eof" (aref *mapline* J))
           (do ((I 0 (1+ I)))
             ((= *map-width* I))              ; Initialize the bottom
             (process-char #\x I J)))         ; "buffer zone" row
       (cond ((>= J *map-length*)
                (princ '|WARNING: Map too long, will be truncated|)(linefeed)
                (process-line (aref *mapline* *map-length*) 1 *map-length*))
             (t
                (princ "   Processing Map Row ")(prin1 J)(linefeed)
                (process-line (aref *mapline* J) 1 J))))
    (princ "Finished Initializing Map")(linefeed)(linefeed))



;*********************************************************************
;***** Function "process-line" cycles thru each character of the
;*****      argument (a string) up to the max allowed width of the map.
;*****      It processes each character and sends a warning
;*****      message to the screen if line is too long.
;*****    Arguments: Line, a string
;*****    Returned: t in all cases.
;*****    Side Effects: Sets a 1-pixel border in right & left columns
```

```
;*****     represented as an obstacle.
;*****   Functions Used: process-char
(defun process-line (Line X Y)
  (cond ((= 0 (length Line))                    ; Normal base case
         (process-char #\x 0 Y)                 ; Initialize the right and left
         (process-char #\x X Y))                ; "buffer zone" columns
        (t (process-char (character (subseq Line 0 1)) X Y)
           (setq X (1+ X))
           (cond ((>= X *map-width*)         ; Abnormal case if map is too wide
                  (process-char #\x 0 Y)       ; Initialize the right and left
                  (process-char #\x (1- X) Y) ; "buffer zone" columns
                  (princ '|WARNING: Map too wide, will be truncated|)(linefeed))
                 (t (process-line (subseq Line 1) X Y))))))

;*****************************************************************
;***** Function "process-char" decodes each character of the map,
;*****     setting the cost element and in some cases the parent
;*****     of the cell indexed by X and Y, the cell's coords,
;*****     and the parent of the cell on the Optimal-Path-List
;*****   Arguments: Char, a character, and X & Y, integers.
;*****   Returned: not applicable.
;*****   Side Effects: Sets the values of the (X,Y,0) element of the
;*****     *cell* array to the cost as specified by the character;
;*****     in some cases sets the values of the (X,Y,1) and (X,Y,3)
;*****     elements for cells having no parent.
;*****   Functions Used: no user-defined functions.
(defun process-char (Char X Y
  &aux X- X+ Y- Y+)
  (setq X- (- X (/ 1 *magnification*)))
  (setq X+ (+ X (/ 1 *magnification*)))
  (setq Y- (- Y (/ 1 *magnification*)))
  (setq Y+ (+ Y (/ 1 *magnification*)))
  (cond ((and (>= (char-int Char) 49) (<= (char-int Char) 57))
         (setf (aref *cell* X Y 0) (- (char-int Char) 48))
         (setf (aref *cell* X Y 1) nil)
         (setf (aref *cell* X Y 2) (cons Char (aref *cell* X Y 2)))
         (setf (aref *cell* X Y 3) nil)
         (cond ((= 1 (aref *cell* X Y 0)) nil)
               (t (setq *terrain-pixel-list*
                    (append
                      (mapcar 'magnify-pixel
                              (list
                                (list X Y)))
                      *terrain-pixel-list*)))))
        ((equal Char #\ )
         (setf (aref *cell* X Y 0) nil)
         (setf (aref *cell* X Y 1) (list X Y))
         (setf (aref *cell* X Y 2) (cons Char (aref *cell* X Y 2)))
         (setf (aref *cell* X Y 3) (list X Y)))
        ((equal Char #\x)
         (setf (aref *cell* X Y 0) nil)
         (setf (aref *cell* X Y 1) (list X Y))
         (setf (aref *cell* X Y 2) (cons Char (aref *cell* X Y 2)))
         (setf (aref *cell* X Y 3) (list X Y))
         (setq X- (- X (/ 1 *magnification*)))
         (setq X+ (+ X (/ 1 *magnification*)))
         (setq Y- (- Y (/ 1 *magnification*)))
         (setq Y+ (+ Y (/ 1 *magnification*)))
         (setq *terrain-pixel-list*
               (append
                 (mapcar 'magnify-pixel
                   (list (list X- Y-)
                         (list X- Y)
                         (list X- Y+)
```

```
                    (list X Y-)
                    (list X Y)
                    (list X Y+)
                    (list X+ Y-)
                    (list X+ Y)
                    (list X+ Y+)))
                *terrain-pixel-list*)))
      ((equal Char #\r)
         (setf (aref *cell* X Y 0) *river-cost*)
         (setf (aref *cell* X Y 1) nil)
         (setf (aref *cell* X Y 2) (cons Char (aref *cell* X Y 2)))
         (setf (aref *cell* X Y 3) nil)
         (setq X+ (+ X (/ 1 *magnification*)))
         (setq Y- (- Y (/ 1 *magnification*)))
         (setq Y+ (+ Y (/ 1 *magnification*)))
         (setq *terrain-pixel-list*
               (append
                 (mapcar 'magnify-pixel
                   (list (list X+ Y-)
                         (list X+ Y)
                         (list X+ Y+)))
                 *terrain-pixel-list*)))
      ((equal Char #\p)
         (setf (aref *cell* X Y 0) *road-cost*)
         (setf (aref *cell* X Y 1) nil)
         (setf (aref *cell* X Y 2) (cons Char (aref *cell* X Y 2)))
         (setf (aref *cell* X Y 3) nil)
         (setq *terrain-pixel-list*
               (cons (list X Y) *terrain-pixel-list*)))
      ((equal Char #\G)
         (setq *goal* (list X Y))
         (setf (aref *cell* X Y 0) 1)
         (setf (aref *cell* X Y 1) (list X Y))
         (setf (aref *cell* X Y 2) (cons Char (aref *cell* X Y 2)))
         (setf (aref *cell* X Y 3) (list X Y)))))


;*****************************************************************************
;*****  Function "process-vertex-info" puts the character v into each
;*****    *cell* X Y 2 as a list (#\v). This becomes the third element of
;*****    this list after "process-edge-info" and "process-char" happen.
(defun process-vertex-info (v-list)
  (setf (aref *cell* (caar v-list) (cadar v-list) 2)
        (list #\v))
  (cond ((null (cdr v-list)))
        (t (process-vertex-info (cdr v-list)))))


;*****************************************************************************
;*****  Function "process-edge-info" puts the id number of the appropriate
;*****    edge into *cell* X Y 2 as the first element of the list there.
;*****    This becomes the second element of the list after "process-char"
;*****    is executed.
;*****    "e-list" is a list of triples: e.g.,
;*****    ((X Y 13) (U V 21) ... (Z W 2)), where for example, 13 is the
;*****    id number of the edge on which cell (X Y) is located.
(defun process-edge-info (e-list)
  (let ((X (first (first e-list)))
        (Y (second (first e-list)))
        (EdgeID (third (first e-list))))
       (cond ((characterp                          ; If cell is a vertex, and
                (first (aref *cell* X Y 2)))       ; no other edge id has been
             (setf (aref *cell* X Y 2)             ; set for this cell, set 1st
                   (cons (list EdgeID)             ; element of list to EdgeID,
```

```
                (aref *cell* X Y 2))))
        ((null (first (aref *cell* X Y 2))) ; If cell is not a vtx, and no
         (setf (aref *cell* X Y 2)          ; other edge id has been set,
                 (list (list EdgeID))))     ; set EdgeID
        ((listp (first (aref *cell* X Y 2)))      ; If another edge id has
         (setf (aref *cell* X Y 2)                ; been set for this cell,
           (cons (cons EdgeID                     ; cons EdgeID onto the
                 (first (aref *cell* X Y 2)))     ; 1st element of the
                 (rest (aref *cell* X Y 2)))))) ; previous list.
(cond ((null (rest e-list)))
      (t (process-edge-info (rest e-list))))))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER -*-

;*****************************************************************************
;*****************************************************************************
;***** File "bdry" contains the functions which detect and record
;***** boundaries between terrain cells.  It also sets and checks
;***** equivalence between optimal-path lists.
;*****
;***** Current as of 27 Jun 89
;*****
;*****************************************************************************
;*****************************************************************************

;*****************************************************************************
;***** Function "check-all-boundaries" iterates through the whole map
;*****   to find boundaries. It is used when boundary-checking is done
;*****   after completion of wavefront expansion.
;*****   Arguments: None
;*****   Returned: not applicable
(defun check-all-boundaries ()
  (setf *boundary-pixel-list* nil)(linefeed)
  (cond
   ((equal "pure" *version*)
    (do ((J 1 (1+ J)))
        ((string-equal "eof" (aref *mapline* J))
         (linefeed)(princ "Finished With Boundary Detection")(linefeed)(linefeed))
        (princ "Pure Bdry Detection for Row ") (prin1 J)(linefeed)
        (do ((I 1 (1+ I)))
            ((>= I *map-width*))
            (cond ((null (aref *cell* (1+ I) J 1)))  ; Check (I,J) against
                  ((pure-bdry-condition              ; (I+1,J)
                         I J (1+ I) J)
                  (add-to-bdry I J (1+ I) J)))
            (cond ((null (aref *cell* I (1+ J) 1)))  ; Check (I,J) against
                  ((pure-bdry-condition              ; (I,J+1)
                          I J I (1+ J))
                  (add-to-bdry I J I (1+ J))))
            (cond ((null (aref *cell* (1+ I) (1+ J) 1))) ; Check (I,J) against
                  ((pure-bdry-condition                  ; (I+1,J+1)
                         I J (1+ I) (1+ J))
                  (add-to-bdry I J (1+ I) (1+ J)))))))))
   ((equal "diverging-path" *version*)
    (do ((J 1 (1+ J)))
        ((string-equal "eof" (aref *mapline* J))
         (linefeed)(princ "Finished With Boundary Detection")(linefeed)(linefeed))
        (princ "Diverging-Path Bdry Detection for Row ") (prin1 J)(linefeed)
        (do ((I 1 (1+ I)))
            ((>= I *map-width*))
            (cond ((null (aref *cell* (1+ I) J 1)))  ; Check (I,J) against
                  ((diverging-path-bdry-condition            ; (I+1,J)
                         I J (1+ I) J)
                  (add-to-bdry I J (1+ I) J)))
            (cond ((null (aref *cell* I (1+ J) 1)))  ; Check (I,J) against
                  ((diverging-path-bdry-condition            ; (I,J+1)
                          I J I (1+ J))
                  (add-to-bdry I J I (1+ J))))
            (cond ((null (aref *cell* (1+ I) (1+ J) 1))) ; Check (I,J) against
                  ((diverging-path-bdry-condition                ; (I+1,J+1)
                         I J (1+ I) (1+ J))
                  (add-to-bdry I J (1+ I) (1+ J)))))))))
   ((equal "vertex-edge" *version*)
    (do ((J 1 (1+ J)))
        ((string-equal "eof" (aref *mapline* J))
         (linefeed)(princ "Finished With Boundary Detection")(linefeed)(linefeed))
```

```
        (princ "Vertex-Edge Bdry Detection for Row ") (prinl J)(linefeed)
        (do ((I 1 (1+ I)))
            ((>= I *map-width*))
            (cond ((null (aref *cell* (1+ I) J 1)))  ; Check (I,J) against
                  ((vertex-edge-bdry-condition       ; (I+1,J)
                         I J (1+ I) J 0 (list I J (1+ I) J))
                  (add-to-bdry I J (1+ I) J)))
            (cond ((null (aref *cell* I (1+ J) 1)))  ; Check (I,J) against
                  ((vertex-edge-bdry-condition       ; (I,J+1)
                          I J I (1+ J) 0 (list I J I (1+ J)))
                  (add-to-bdry I J I (1+ J))))
            (cond ((null (aref *cell* (1+ I) (1+ J) 1))) ; Check (I,J) against
                  ((vertex-edge-bdry-condition          ; (I+1,J+1)
                         I J (1+ I) (1+ J) 0 (list I J (1+ I) (1+ J)))
                  (add-to-bdry I J (1+ I) (1+ J)))))))))


;*************************************************************************
;***** Function "check-for-boundaries" checks each of a cell's four orthogonal
;*****   neighbors for the existence of a region boundary. It is used
;*****   when boundary-checking is done incrementally during wave expansion.
;*****   Arguments: Center-cell,a list of the coords of the cell being
;*****              checked and aux (local) variables to hold the coords
;*****   Returned: not applicable
;*****   Side Effects: if bdry exists, the appropriate pixels are added
;*****     to *boundary-pixel-list* and *boundary-bit*(X,Y) is set.
;*****   Functions Used: check-neighbor
;*****
(defun check-for-boundaries (Center-cell
         &aux X Y)
  (setq X (car Center-cell))
  (setq Y (cadr Center-cell))
  (cond
   ((equal "vertex-edge" *version*)
     (cond ((null (aref *cell* X (1- Y) 1)))   ; Check (X,Y) against (X,Y-1)
                                               ; If (X,Y-1)'s parent is undefined
                                               ; boundary cannot be checked yet,
           ((vertex-edge-bdry-condition     ; Else it can so call bdry condition.
              X Y X (1- Y) 0 (list X Y X (1- Y)))
            (add-to-bdry X Y X (1- Y))))            ; If bdry-cond = T, add to bdry-list.
     (cond ((null (aref *cell* (1- X) Y 1)))
           ((vertex-edge-bdry-condition     ; Check (X,Y) against (X-1,Y)
              X Y (1- X) Y 0 (list X Y (1- X) Y))
            (add-to-bdry X Y (1- X) Y)))
     (cond ((null (aref *cell* X (1+ Y) 1)))
           ((vertex-edge-bdry-condition     ; Check (X,Y) against (X,Y+1)
              X Y X (1+ Y) 0 (list X Y X (1+ Y)))
            (add-to-bdry X Y X (1+ Y))))
     (cond ((null (aref *cell* (1+ X) Y 1)))
           ((vertex-edge-bdry-condition     ; Check (X,Y) against (X+1,Y)
              X Y (1+ X) Y 0 (list X Y (1+ X) Y))
            (add-to-bdry X Y (1+ X) Y))))
   ((equal "pure" *version*)
     (cond ((null (aref *cell* X (1- Y) 1)))   ; Check (X,Y) against (X,Y-1)
                                               ; If (X,Y-1)'s parent is undefined
                                               ; boundary cannot be checked yet,
           ((pure-bdry-condition            ; Else it can so call bdry condition.
              X Y X (1- Y))
            (add-to-bdry X Y X (1- Y))))            ; If bdry-cond = T, add to bdry-list.
     (cond ((null (aref *cell* (1- X) Y 1)))
           ((pure-bdry-condition            ; Check (X,Y) against (X-1,Y)
              X Y (1- X) Y)
            (add-to-bdry X Y (1- X) Y)))
```

```
    (cond ((null (aref *cell* X (1+ Y) 1)))
          ((pure-bdry-condition            ; Check (X,Y) against (X,Y+1)
             X Y X (1+ Y))
           (add-to-bdry X Y X (1+ Y))))
    (cond ((null (aref *cell* (1+ X) Y 1)))
          ((pure-bdry-condition            ; Check (X,Y) against (X+1,Y)
             X Y (1+ X) Y)
           (add-to-bdry X Y (1+ X) Y))))
  ((equal "diverging-path" *version*)
    (cond ((null (aref *cell* X (1- Y) 1)))   ; Check (X,Y) against (X,Y-1)
                                              ; If (X,Y-1)'s parent is undefined
                                              ; boundary cannot be checked yet,
          ((diverging-path-bdry-condition         ; Else it can so call bdry condition.
             X Y X (1- Y))
           (add-to-bdry X Y X (1- Y))))           ; If bdry-cond = T, add to bdry-list.
    (cond ((null (aref *cell* (1- X) Y 1)))
          ((diverging-path-bdry-condition         ; Check (X,Y) against (X-1,Y)
             X Y (1- X) Y)
           (add-to-bdry X Y (1- X) Y)))
    (cond ((null (aref *cell* X (1+ Y) 1)))
          ((diverging-path-bdry-condition         ; Check (X,Y) against (X,Y+1)
             X Y X (1+ Y))
           (add-to-bdry X Y X (1+ Y))))
    (cond ((null (aref *cell* (1+ X) Y 1)))
          ((diverging-path-bdry-condition         ; Check (X,Y) against (X+1,Y)
             X Y (1+ X) Y)
           (add-to-bdry X Y (1+ X) Y))))))



;*************************************************************************
;***** Function "vertex-edge-bdry-condition" checks if there is a boundary
;*****  between two cells by seeing if their OPL's have equivalent "critical"
;*****  points, where a critical point is a turn-cell which is on an edge
;*****  or is adjacent to a terrain-feature vertex.
;*****  Arguments: Coords of 2 cells which may be in different regions;
;*****    Flag which is 0 normally, but for double-edged cells on the
;*****    second recursive call with that cell is a list of the left-over edge-id,
;*****    and for edge-interior pairs is 1 or (edge-id) after initial call.
;*****  Returned: nil if condition does not hold, and T
;*****    if condition does hold.
;*****  Side Effects: none
;*****
(defun vertex-edge-bdry-condition (X1 Y1 X2 Y2 Flag StartPoints)
  (let* ((Xs1 (first StartPoints))              ; bdry cond based on vertex
         (Ys1 (second StartPoints))             ; and edge turn points.
         (Xs2 (third StartPoints))
         (Ys2 (fourth StartPoints))
         (Parent1 (first-distinguished-opl-cell X1 Y1 Xs1 Ys1))
         (Parent2 (first-distinguished-opl-cell X2 Y2 Xs2 Ys2))
         (Xp1 (first Parent1))
         (Yp1 (second Parent1))
         (Xp2 (first Parent2))
         (Yp2 (second Parent2)))
    (cond
      ((not (equal                    ; Case A: If the start-pts themselves have
         (first (aref *cell* Xs1 Ys1 2))    ; different costs, they are in different
         (first (aref *cell* Xs2 Ys2 2))))  ; regions. This condition fires only
       't)                                  ; on the 1st call to v-e-b-c.
      ((and                         ; Case B1: If SP1 is edge (& SP2 is inside same
         (< 1 (length (aref *cell* Xs1 Ys1 2)))  ; rgn, by A above) and Parent of 2
         (equal                                  ; is on the same edge as SP1, and
           (second (aref *cell* Xs1 Ys1 2))      ; SP1 is not the 1st of a pair of
           (third Parent2))                      ; double-edge cells, do not put a
```

```
      (not (equal                                  ; bdry between SP1 & SP2.
        (second (aref *cell* Xs1 Ys1 2))           ; (This case makes edge cells &
        (third Parent1))))                         ; interior cells be in same rgn.)
     nil)
    ((and                          ; Case B2: If SP2 is edge (& SP1 is inside same
      (< 1 (length (aref *cell* Xs2 Ys2 2)))       ; rgn, by A above) and Parent of 1
      (equal                                       ; is on the same edge as SP2, and
        (second (aref *cell* Xs2 Ys2 2))           ; SP2 is not the 1st of a pair of
        (third Parent1))                           ; double-edge cells, do not put a
      (not (equal                                  ; bdry between SP1 & SP2.
        (second (aref *cell* Xs2 Ys2 2))           ; (This case makes edge cells &
        (third Parent2))))                         ; interior cells be in same rgn.)
     nil)
    ((and (= Xp1 Xp2)              ; Case C: If Parent1 & Parent2 are the same,
          (= Yp1 Yp2)) nil)           ; Pt1 & Pt2 are in same region.
    ((and (= 3 (length Parent1))
          (= 3 (length Parent2))) ; Case D: If parents are both edge cells:
     (cond
       ((set-equal (third Parent1)   ; Case D1:    If edge-id lists are the same,
                   (third Parent2))    ; chk next pair of cells on OPL recursively
        (vertex-edge-bdry-condition    ; (Normal case)
                 Xp1 Yp1 Xp2 Yp2 0 StartPoints))
       ((and                          ; Case D2:
          (subsetp (third Parent2)      ; Else if Parent1 is a double-edge cell and
                   (third Parent1))     ; one of its edge-ids = edge-id of Parent2,
          (not (listp Flag)))           ; and this is the 1st time Parent1 has been
        (vertex-edge-bdry-condition     ; checked in this set of calls to v-e-b-c
              X1 Y1 Xp2 Yp2             ; recursively check OPL with Point1 and
              (set-difference           ; Parent2, with flag := (unmatched-edge-id)
                (third Parent1)         ; of Parent1. (Only applies where cell 1 is
                (third Parent2))        ; is on two edges.)
              StartPoints))
       ((and                          ; Case D3;
          (subsetp (third Parent1)      ; Else if Parent2 is a double-edge cell and
                   (third Parent2))     ; one of its edge-ids = edge-id of Parent1,
          (not (listp Flag)))           ; and this is the 1st time Parent2 has been
        (vertex-edge-bdry-condition     ; checked in this set of calls to
              Xp1 Yp1 X2 Y2             ; v-e-bdry-cond, recursively check OPL with
              (set-difference           ; Parent1 and Point2, with flag :=
                (third Parent2)         ; (unmatched-edge-id) of Parent2.
                (third Parent1))
              StartPoints))
       ((and                          ; Case D2, Second Pass:
          (subsetp (third Parent2)      ; Else if Parent1 is a double-edge cell and
                   (third Parent1))     ; its previously unmatched edge-id = id of
          (equal Flag                   ; Parent2, recursively check OPL from
                 (third Parent2)))      ; Parent1 & Parent2, with Flag = NIL.
        (vertex-edge-bdry-condition
             Xp1 Yp1 Xp2 Yp2 0 StartPoints))
       ((and                          ; Case D3, Second Pass:
          (subsetp (third Parent1)      ; Else if Parent2 is a double-edge cell and
                   (third Parent2))     ; and previously unmatched edge-id = id of
          (equal Flag                   ; Parent1, recursively check OPL from
                 (third Parent1)))      ; Parent1 and Parent2, with Flag = NIL.
        (vertex-edge-bdry-condition
             Xp1 Yp1 Xp2 Yp2 0 StartPoints))
       (t 't)))                       ; Case D4: Otherwise pts are in different rgns
     (t 't))))                        ; Case E: OTHERWISE pts are in different rgns.

;*************************************************************************
;***** Function "first-distinguished-opl-cell" finds the first cell on
;*****   the opl of Pt X,Y which is a "distinguished" point. It is called
;*****   by function "heuristic-bdry-condition"
```

```
;*****   Arguments: coords of the cell whose opl is being checked
;*****   Returned: a list of the coordinates of the distinguished cell,
;*****     followed if it is an edge cell by the edge id num.
;*****   Side effects: none
(defun first-distinguished-opl-cell (X Y Xs Ys
  &aux Dcell)
  (cond ((equalp                                ; If opl-parent=point, cell is obstacle
           (list X Y)                           ; or goal, so return the point itself.
           (aref *cell* X Y 3)) (list X Y))     ; (base case 2)
        ((setf Dcell (distinguished-cell        ; If opl-parent is distinguished, rtn
           (first (aref *cell* X Y 3))          ; coords of parent and possibly the
           (second (aref *cell* X Y 3))))       ; edge id number. (base case 3)
         (cond
           ((= 3 (length Dcell)) Dcell)         ; If Dcell is edge cell, rtn Dcell.
           ((not (equal                         ; If Dcell is vertex and this path
              (first (aref *cell* Xs Ys 2))     ; started outside the terrain feature
              (first (aref *cell*               ; of which Dcell is a vtx, rtn Dcell.
                       (first Dcell)
                       (second Dcell)
                       2))))
            Dcell)
           (t (first-distinguished-opl-cell         ; Else, recurse to look
                (first (aref *cell* X Y 3))         ; at next cell on opl.
                (second (aref *cell* X Y 3))
                Xs Ys))))
        (t (first-distinguished-opl-cell          ; Else, recurse to look
             (first (aref *cell* X Y 3))          ; at next cell on opl.
             (second (aref *cell* X Y 3))
             Xs Ys))))

;*********************************************************************
;***** Function "distinguished-cell" determines whether cell is an edge
;*****   or adjacent to a terrain-feature vertex.
;*****   Arguments: coords of the cell being checked for disting. status
;*****   Returned: (X Y edge-id-list) if cell is on an edge
;*****             (X Y) if cell is adjacent to a vertex
;*****             nil if cell is not distinguished
;*****   Side effects: none
(defun distinguished-cell (X Y)
  (let ((X- (1- X))
        (X+ (1+ X))
        (Y- (1- Y))
        (Y+ (1+ Y)))
    (cond
      ((and (< 1 (length (aref *cell* X Y 2)))          ; If (X,Y) is edge cell
            (equal                                      ; and is the first of a
              (second (aref *cell* X Y 2))              ; pair of adjacent cells
              (second                                   ; of the same edge in the
                (aref *cell*                            ; same backpath,
                  (first (aref *cell* X Y 1))           ; return nil.
                  (second (aref *cell* X Y 1))
                  2)))) nil)
      ((< 1 (length (aref *cell* X Y 2)))           ; If (X,Y) is a single edge cell,
       (list X Y (second (aref *cell* X Y 2))))     ; return coords & edge-id-list.
      ((and (= 3 (length (aref *cell* X- Y 2)))
            (not (equalp (first (aref *cell* X Y 2))        ; Else if (X,Y) is
                         (first (aref *cell* X- Y 2)))))    ; adjacent to a vertex
       (list X- Y))                                         ; and is outside the
      ((and (= 3 (length (aref *cell* X- Y+ 2)))            ; terrain feature of
            (not (equalp (first (aref *cell* X Y 2))        ; which the vertex is
                         (first (aref *cell* X- Y+ 2)))))   ; a part, return coords
       (list X- Y+))                                        ; of vertex.
      ((and (= 3 (length (aref *cell* X Y- 2)))
            (not (equalp (first (aref *cell* X Y 2))
```

```
                        (first (aref *cell* X Y- 2)))))
       (list X Y-))
      ((and (= 3 (length (aref *cell* X- Y- 2)))
            (not (equalp (first (aref *cell* X Y 2))
                         (first (aref *cell* X- Y- 2)))))
       (list X- Y-))
      ((and (= 3 (length (aref *cell* X Y+ 2)))
            (not (equalp (first (aref *cell* X Y 2))
                         (first (aref *cell* X Y+ 2)))))
       (list X Y+))
      ((and (= 3 (length (aref *cell* X+ Y- 2)))
            (not (equalp (first (aref *cell* X Y 2))
                         (first (aref *cell* X+ Y- 2)))))
       (list X+ Y-))
      ((and (= 3 (length (aref *cell* X+ Y 2)))
            (not (equalp (first (aref *cell* X Y 2))
                         (first (aref *cell* X+ Y 2)))))
       (list X+ Y))
      ((and (= 3 (length (aref *cell* X+ Y+ 2)))
            (not (equalp (first (aref *cell* X Y 2))
                         (first (aref *cell* X+ Y+ 2)))))
       (list X+ Y+))
      (t nil))))                          ; Else (X,Y) is not adjacent to a vertex
                                          ; and is not an edge cell


;*****************************************************************************
;***** Function "add-to-bdry" sets the boundary bit to 1 and
;*****   adds boundary pixels to the front of the boundary list
;*****   unless one of the arguments is an obstacle cell.
;*****   Argument: coords of two points whose boundary
;*****     is to be added.
;*****   Returned: always returns T
;*****   Side effects: Sets *boundary* bit to 1 and
;*****     sets *boundary-pixel-list* to the previous
;*****     list with the new pixels appended to the front.
;*****
(defun add-to-bdry (X1 Y1 X2 Y2
  &aux Xa Xb Ya Yb)
  (cond
    ((or (char-equal #\x (car (aref *cell* X1 Y1 2)))
         (char-equal #\x (car (aref *cell* X2 Y2 2))))'t)
    (t
      (setq Xa (+ X1 (/ (- X2 X1) *magnification*)))
      (setq Ya (+ Y1 (/ (- Y2 Y1) *magnification*)))
      (setq Xb (+ X1 (* 2 (/ (- X2 X1) *magnification*))))
      (setq Yb (+ Y1 (* 2 (/ (- Y2 Y1) *magnification*))))
      (setf (bit *boundary*
                 (min X1 X2)                  ; Set the boundary
                 (min Y1 Y2)                  ; flag bit of the upper
                 (cond ((= 0 (- Y1 Y2)) 0)    ; or leftmost cell.
                       (t 1)))
            1)
      (setf *boundary-pixel-list*
            (append
              (mapcar 'magnify-pixel
                (cond
                  ((= X1 X2)
                   (list (list (- X1 (/ 1 *magnification*)) Ya)
                         (list            X1                Ya)
                         (list (+ X1 (/ 1 *magnification*)) Ya)
                         (list (- X1 (/ 1 *magnification*)) Yb)
                         (list            X1                Yb)
```

```
                    (list (+ X1 (/ 1 *magnification*)) Yb)))
               ((= Y1 Y2)
                (list (list Xa (- Y1 (/ 1 *magnification*)))
                    (list Xa                Y1            )
                    (list Xa (+ Y1 (/ 1 *magnification*)))
                    (list Xb (- Y1 (/ 1 *magnification*)))
                    (list Xb                Y1            )
                    (list Xb (+ Y1 (/ 1 *magnification*)))))
               (t
                (list (list Xa Ya)
                    (list Xa Yb)
                    (list Xb Ya)
                    (list Xb Yb)))))
          *boundary-pixel-list*)) 't)))

;*********************************************************************
;***** Function "set-opl" sets the coords for a cell's predecessor
;*****   in the optimal-path-list.
;*****   Arguments: Xn and Yn the coords of the new cell with OPL being
;*****     set and Xp and Yp the coords of (Xn,Yn)'s parent on backpath
;*****   Returned: not applicable
;*****   Side Effects: sets *cell*(Xn,Yn,3) with n's predecessor on OPL
;*****   Functions Used: on-line-between
;*****
(defun set-opl (Xn Yn Xp Yp)
   (cond ((< 1 (length (aref *cell* Xp Yp 2)))        ; If P is an edge cell, set
          (setf (aref *cell* Xn Yn 3) (list Xp Yp)))  ; pred of N to P.
         ((on-line-between Xp Yp Xn Yn                    ; If P is between N
                           (first (aref *cell* Xp Yp 3))  ; & pred of P on OPL,
                           (second (aref *cell* Xp Yp 3))) ; set pred of N to
          (setf (aref *cell* Xn Yn 3) (aref *cell* Xp Yp 3)))     ; pred of P.
         (t (setf (aref *cell* Xn Yn 3) (list Xp Yp)))))   ; Else set pred of N
                                                           ; to P itself.


;*********************************************************************
;***** Function "pure-bdry-condition"
;*****  Arguments: Coords of 2 cells which may be in different regions
;*****  Returned: nil if condition does not hold, and T
;*****    if condition does hold.
;*****  Side Effects: none
;*****
(defun pure-bdry-condition (X1 Y1 X2 Y2)          ; "Pure" boundary condition:
  (let* ((Xp1 (car (aref *cell* X1 Y1 3)))     ; If OPLs are equivalent,
         (Yp1 (cadr (aref *cell* X1 Y1 3)))    ; return nil, else return T
         (Xp2 (car (aref *cell* X2 Y2 3)))
         (Yp2 (cadr (aref *cell* X2 Y2 3)))
         (Xpp1 (car (aref *cell* Xp1 Yp1 3)))
         (Ypp1 (cadr (aref *cell* Xp1 Yp1 3)))
         (Xpp2 (car (aref *cell* Xp2 Yp2 3)))        ; OPLs are equivalent if
         (Ypp2 (cadr (aref *cell* Xp2 Yp2 3))))      ; first cells in each
        (cond ((and (= Xp1 Xp2)                      ; OPL are equivalent, ie,
                    (= Yp1 Yp2)) nil)                ; if they are the same,
                                                     ; or if one is in the
              ((on-line-incl-between                 ; first leg of the
                   Xp1 Yp1 Xp2 Yp2 Xpp2 Ypp2) nil)   ; OPL of the other
              ((on-line-incl-between
                   Xp2 Yp2 Xp1 Yp1 Xpp1 Ypp1) nil)
              ((on-line-incl-between
                   Xp1 Yp1 X2 Y2 Xp2 Yp2) nil)
              ((on-line-incl-between
                   Xp2 Yp2 X1 Y1 Xp1 Yp1) nil)
              (t (add-to-bdry X1 Y1 X2 Y2))))))
```

```
;*************************************************************************
;***** Function "diverging-path-bdry-condition"
;*****  Arguments: Coords of 2 cells which may be in different regions
;*****  Returned: nil if condition does not hold, and T
;*****    if condition does hold.
;*****  Side Effects: none
;*****
(defun diverging-path-bdry-condition (Xc Yc Xn Yn
  &aux PC PXc PYc PPc PPXc PPYc PPPC PPPXc PPPYc PN PXn PYn PPn PPXn PPYn PPPN PPPXn PPPYn
  (setq PC (aref *cell* Xc Yc 1))
  (setq PN (aref *cell* Xn Yn 1))
  (setq PXc (first PC))                    ; Find Center-cell's
  (setq PYc (second PC))                   ; parent, grandparent, and
  (setf PPC (aref *cell* PXc PYc 1))       ; great-grandparent
  (setq PPXc (first PPC))
  (setq PPYc (second PPC))
  (setf PPPC (aref *cell* PPXc PPYc 1))
  (setq PPPXc (first PPPC))
  (setq PPPYc (second PPPC))
  (setq PXn (first PN))                    ; Find Neighbor-cell's
  (setq PYn (second PN))                   ; parent, grandparent, and
  (setf PPN (aref *cell* PXn PYn 1))       ; great-grandparent
  (setq PPXn (first PPN))
  (setq PPYn (second PPN))
  (setf PPPN (aref *cell* PPXn PPYn 1))
  (setq PPPXn (first PPPN))
  (setq PPPYn (second PPPN))
  (cond ((and (= Xn PXn) (= Yn PYn))) ; Keeps obst from causing bdry.
        ((bdry-condition-1                 ; If greatgp's are separated
           PPPXc PPPYc PPPXn PPPYn)        ; by more than two, cells are
         (add-to-bdry Xc Yc Xn Yn))        ; in different regions.
        ((bdry-condition-2                   ; If cell and neighbor's
           Xc Yc PXn PYn)                    ; parent are in different
         (add-to-bdry Xc Yc Xn Yn))          ; regions, so are cells.
        ((bdry-condition-2                     ; If parents are in
           PXc PYc PXn PYn)                    ; different regions,
         (add-to-bdry Xc Yc Xn Yn))            ; so are cells.
        ((bdry-condition-3                 ; If gp's are separated by an
           PPXc PPYc PPXn PPYn)            ; obst or river cell, there
         (add-to-bdry Xc Yc Xn Yn))        ; is a bdry btwn cells.
        (t nil)))

;*************************************************************************
;***** Function "bdry-condition-1"
;*****  Arguments: Coords of 2 cells which may be in different regions
;*****  Returned: nil if condition does not hold, and T
;*****    if condition does hold.
;*****  Side Effects: none
;*****
(defun bdry-condition-1 (X1 Y1 X2 Y2)
       (cond                          ; Boundary condition 1:
         ((or (< 2 (abs (- X1 X2)))       ; If cells are more than 2 cells
              (< 2 (abs (- Y1 Y2)))))     ; apart, return "true"
         (t nil)))

;*************************************************************************
;***** Function "bdry-condition-2"
;*****  Arguments: Coords of 2 cells which may be in different regions
;*****  Returned: nil if condition does not hold, and T if
;*****    condition does hold.
;*****  Side Effects: none
;*****
(defun bdry-condition-2 (X1 Y1 X2 Y2)
```

```
        (cond                                   ; Boundary condition 2:
         ((and
            (= 1 (+ (abs (- X1 X2))                 ; If cells are adjacent,
                    (abs (- Y1 Y2))))
            (not (equal (list X1 Y1)                ; and one is not the
                        (aref *cell* X2 Y2 1)))     ; parent of the other,
            (not (equal (list X2 Y2)
                        (aref *cell* X1 Y1 1)))
            (= 1 (bit *boundary*                    ; and bdry bit is set,
                        (min X1 X2)
                        (min Y1 Y2)
                        (cond ((= 0 (- Y1 Y2)) 0)
                              (t 1)))))
          't)                                       ; then return "true"
         (t nil)))                                  ; else return "nil"

;*********************************************************************
;***** Function "bdry-condition-3" checks if points are separated
;*****  by exactly one obstacle or river cell.  If so, under the
;*****  circumstances in which cond-3 will be called, they are in
;*****  different regions.
;*****  Arguments: Coords of 2 cells which may be in different regions
;*****  Returned: nil if condition does not hold, and T
;*****    if condition does hold.
;*****  Side Effects: none
;*****  NOTE: nested cond's are arranged as they are to detect as soon
;*****   as possible when the conditions will not hold, because this test
;*****   must be run 4 times for every cell in the map, and only occasionally
;*****   will the =2,=0 conditions be true.
;*****
(defun bdry-condition-3 (X1 Y1 X2 Y2)    ; If cells are  2 apart horizontally,
       (cond                             ; and 0 apart vertically, and
         ((and (= 2 (abs (- X1 X2)))     ; if cell between them is an obstacle
               (= 0 (- Y1 Y2)))          ; or river, their children are in
          (cond                          ; different regions.
            ((or (char-equal #\x (aref *cell* (/ (+ X1 X2) 2) Y1 2))
                 (char-equal #\r (aref *cell* (/ (+ X1 X2) 2) Y1 2)))
             't)))
         ((and (= 0 (- X1 X2))           ; Same as above for 2 apart vertically
               (= 2 (abs (- Y1 Y2))))    ; and 0 apart horizontally.
          (cond
            ((or (char-equal #\x (aref *cell* X1 (/ (+ Y1 Y2) 2) 2))
                 (char-equal #\r (aref *cell* X1 (/ (+ Y1 Y2) 2) 2)))
             't)))
         (t nil)))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER -*-

;*****************************************************************************
;*****************************************************************************
;***** "utils.lisp" contains several utility functions used by "opm"
;***** found in file "opm.lisp" and related functions.
;*****
;***** Current as of 7 Jun 89
;*****
;*****************************************************************************
;*****************************************************************************

;*****************************************************************************
;***** Function "on-line-incl-between" determines whether the first point
;*****   is between the second and third, inclusive.
;*****   Arguments: X & Y, X1 & Y1, X2 & Y2, coords of three points
;*****   Returned: non-nil if (X,Y) is strictly btwn (X1,Y1) & (X2,Y2),
;*****     or nil otherwise.
;*****   Side Effects: none
(defun on-line-incl-between (X Y X1 Y1 X2 Y2)
   (cond ((and (= X X1) (= Y Y1)))     ; If (X,Y) = (X1,Y1) or
         ((and (= X X2) (= Y Y2)))     ; (X,Y) = (X2,Y2), return T
         ((and (or (< X1 X X2)         ; If (X,Y) is strictly
                   (> X1 X X2))        ; inside the rectangle
               (or (< Y1 Y Y2)         ; formed by the line
                   (> Y1 Y Y2))        ; endpoints, check by
               (= (/ (- Y Y1)          ; comparing slopes whether
                     (- X X1))         ; point is on line.
                  (/ (- Y Y2)
                     (- X X2)))))
         ((and (= X X1 X2)             ; If line is vertical, check by
               (or (< Y1 Y Y2)         ; comparing Y coordinates.
                   (> Y1 Y Y2))))
         ((and (= Y Y1 Y2)             ; If line is horizontal, check by
               (or (< X1 X X2)         ; comparing X coordinates.
                   (> X1 X X2))))
         (t nil)))                           ; Else return NIL

;*****************************************************************************
;***** Function "on-line-between" determines whether the first point
;*****   is strictly between the second and third.
;*****   Arguments: X & Y, X1 & Y1, X2 & Y2, coords of three points
;*****   Returned: non-nil if (X,Y) is strictly btwn (X1,Y1) & (X2,Y2),
;*****     or nil otherwise.
;*****   Side Effects: none
(defun on-line-between (X Y X1 Y1 X2 Y2)
   (cond ((and (or (< X1 X X2)         ; If (X,Y) is strictly
                   (> X1 X X2))        ; inside the rectangle
               (or (< Y1 Y Y2)         ; formed by the line
                   (> Y1 Y Y2))        ; endpoints, check by
               (= (/ (- Y Y1)          ; comparing slopes whether
                     (- X X1))         ; point is on line.
                  (/ (- Y Y2)
                     (- X X2)))))
         ((and (= X X1 X2)             ; If line is vertical, check by
               (or (< Y1 Y Y2)         ; comparing Y coordinates.
                   (> Y1 Y Y2))))
         ((and (= Y Y1 Y2)             ; If line is horizontal, check by
               (or (< X1 X X2)         ; comparing X coordinates.
                   (> X1 X X2))))
         (t nil)))                           ; Else return NIL

;*****************************************************************************
;***** Function "magnify-pixel" takes a pair of pixel coordinates
```

```
;*****   and returns coordinates which are k times magnified.
;*****   Argument: Pixel, a list of two numbers, and K, the magnification.
;*****   Returned: a list of two numbers, each number
;*****     being K times the original.
;*****   Side Effects: none
(defun magnify-pixel (Pixel)
  (list (* *magnification* (first Pixel))
        (* *magnification* (second Pixel))))

;*************************************************************************
;***** Function "magnify-pixel-list" takes a list of pixel coordinates
;*****   and returns a list which is k times magnified.
;*****   Argument: Pixel-list, a list of lists of two numbers each,
;*****     and K, the magnification.
;*****   Returned: a list of lists of two numbers, each number
;*****     being K times the original.
;*****   Side Effects: none
(defun magnify-pixel-list (Pixel-list)
  (cond ((null Pixel-list) nil)
        (t (cons (list (* *magnification* (first (first Pixel-list)))
                       (* *magnification* (second (first Pixel-list))))
                 (magnify-pixel-list (rest Pixel-list))))))

;*************************************************************************
;***** Function get-backpath finds a cell's parent, and gives the
;*****  pixels from the cell to the parent, including the cell.
;*****  This version only works for *magnif* = 2 or 3
;*****  Argument: X & Y, coords of cell whose backpath is required
;*****  Returned: a list of pixel coords
;*****  Side Effects: none
;*****
(defun get-backpath (X Y
  &aux Parent-cell Xm Ym Xp Yp)
  (cond ((null (aref *cell* X Y 1)) nil)
        (t
           (setq Parent-cell (magnify-pixel (aref *cell* X Y 1)))
           (setq Xp (first Parent-cell))
           (setq Yp (second Parent-cell))
           (setq Xm (* *magnification* X))
           (setq Ym (* *magnification* Y))
           (list (list Xm Ym)
                 (list (+ Xm (/ (- Xp Xm) *magnification*))
                       (+ Ym (/ (- Yp Ym) *magnification*)))
                 (list (+ Xm (* 2 (/ (- Xp Xm) *magnification*)))
                       (+ Ym (* 2 (/ (- Yp Ym) *magnification*)))))))))


;*************************************************************************
;***** Function get-all-backpaths finds the backpaths from every cell
;*****  on the map and puts them in pixel form into *backpath-pixel-list*
(defun get-all-backpaths ()
   (setq *backpath-pixel-list* nil)
   (do ((J 1 (1+ J)))
       ((string-equal "eof" (aref *mapline* J)) *backpath-pixel-list*)
       (do ((I 1 (1+ I)))
           ((= (length (aref *mapline* 1)) I))
           (setq *backpath-pixel-list*
                 (append
                    (get-backpath I J)
                    *backpath-pixel-list*)))))

;*************************************************************************
;***** Function set-equal checks if two sets are the same.
;*****   Arguments: Set1 and Set2, two lists treated as sets.
```

```
;*****   Returned: T if Set1 and Set2 are the same, disregarding
;*****     repeated elements, NIL otherwise.
(defun set-equal (Set1 Set2)
  (cond ((and (subsetp Set1 Set2)
              (subsetp Set2 Set1)))
        (t nil)))

;*************************************************************************
;***** Function print-opl is a debugging function to print the OPL
;*****  of a cell to the screen.
(defun print-opl (X Y)
  (cond ((equal *goal* (list X Y)))
        (t (print (aref *cell* X Y 3))
           (prin1 (aref *cell*
                        (first (aref *cell* X Y 3))
                        (second (aref *cell* X Y 3))
                        2))
           (print-opl (first (aref *cell* X Y 3))
                      (second (aref *cell* X Y 3))))))

;*************************************************************************
;***** Function "linefeed" is a mnemonic for terpri.
;*****   Side Effects: causes a carriage return to be sent to
;*****     the output stream.
(defun linefeed ()
  (terpri))
(defun linefeed2 ()
  (terpri *output-stream*))


;*************************************************************************
;***** Function "report-completion" sends a message to the screen
(defun report-completion ()
  (linefeed)
  (princ "Wavefront expansion complete") (linefeed)
  (princ "Type (kill-windows) to remove screen")(linefeed) 't)


;*************************************************************************
;***** Function sort-condition determines the order between
;*****   two cells on the wavefront.
;*****   Arguments: two sets of coordinates
;*****   Returned: TRUE if remaining cost of first cell is less than
;*****     remaining cost of second.
;*****   Side Effects: none
(defun sort-condition (Cell1 Cell2)
  (let ((X1 (first Cell1))
        (Y1 (second Cell1))
        (X2 (first Cell2))
        (Y2 (second Cell2)))
    (< (aref *cell* X1 Y1 0)
       (aref *cell* X2 Y2 0))))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER -*-
;*****************************************************************************
;*****************************************************************************
;*****  File "graphics" contains the functions to open a window for
;*****  displaying the terrain, wavefront, boundaries, and back-paths.
;*****  It is adapted from file "graph.lisp" written by Dr. Se-Hung Kwak
;*****
;*****  Current as of 13 May 88
;*****
;*****************************************************************************
;*****************************************************************************
(defvar *display-window*)
(defvar *draw-window*)
(defvar *draw-window-array*)
(defvar *draw-window-width*)
(defvar *draw-window-height*)
(defvar *draw-window-inside-width*)
(defvar *draw-window-inside-height*)
(defvar *draw-window-position*)

(defun initialize-graphics ()
  (initialize-windows)
  (clear-window)
  (draw-goal)
  (draw-features *terrain-pixel-list*)
  (make-visible))

(defun draw-and-show-window ()
  (mapcar 'draw-pt *boundary-pixel-list*)
  (mapcar 'draw-pt *backpath-pixel-list*)
  (make-visible)
  (setq *backpath-pixel-list* nil))

(defun draw-and-show-bdry-window ()
  (clear-window)
  (draw-goal)
  (draw-features *terrain-pixel-list*)
  (mapcar 'draw-pt *boundary-pixel-list*)
  (make-visible))

(defun draw-and-show-backpaths ()
  (clear-window)
  (draw-goal)
  (draw-features (get-all-backpaths))
  (make-visible)
  't)

(defun show-boundary ()
  (mapcar 'draw-pt *boundary-pixel-list*)
  (make-visible)'t)

(defun show-terrain ()
  (draw-features *terrain-pixel-list*)
  (make-visible) 't)

(defun show-goal ()
  (draw-goal)
  (make-visible) 't)

(defun show-backpaths ()
  (draw-features (get-all-backpaths))
  (make-visible) 't)

(defun initialize-windows()
```

```
  (princ "Initializing Windows")(linefeed)(linefeed)
  (setf *draw-window-width* 650)
  (setf *draw-window-height* 500)
  (setf *draw-window-position* '(75 75))
  (setf *display-window*
        (tv:make-window 'tv:window
                        :blinker-p nil
                        :position *draw-window-position*
                        :width *draw-window-width*
                        :height *draw-window-height*
                        :name "display-window"
                        :save-bits t
                        :expose-p t))
  (setf *draw-window*
        (tv:make-window 'tv:window
                        :blinker-p nil
                        :position *draw-window-position*
                        :width *draw-window-width*
                        :height *draw-window-height*
                        :name "draw-window"
                        :save-bits t
                        :expose-p nil))
    (setf *draw-window-array*
        (send *draw-window* :bit-array))
    (setf *draw-window-inside-height*
          (send *draw-window* :inside-height))
    (setf *draw-window-inside-width*
          (send *draw-window* :inside-width)))

(defun clear-window()
  (tv:sheet-force-access (*draw-window*)
    (send *draw-window* :refresh)))


(defun draw-goal()
  (tv:sheet-force-access (*draw-window*)
    (send *draw-window* :draw-string "*"
          (- (* *magnification* (first *goal*)) 3)
          (+ (* *magnification* (second *goal*)) 5))))

(defun make-visible()
  (send *display-window* :bitblt
        tv:alu-seta
        *draw-window-inside-width*
        *draw-window-inside-height*
        *draw-window-array*
        2 2 0 0))

(defun kill-windows()
  (send *display-window* :kill)
  (send *draw-window* :kill)
  (linefeed)(princ "Windows Killed")(linefeed)(linefeed))

(defun draw-features (Pixel-list)
  (do ((Rest-of-list Pixel-list (rest Rest-of-list)))
      ((null Rest-of-list))
      (draw-pt (first Rest-of-list))))

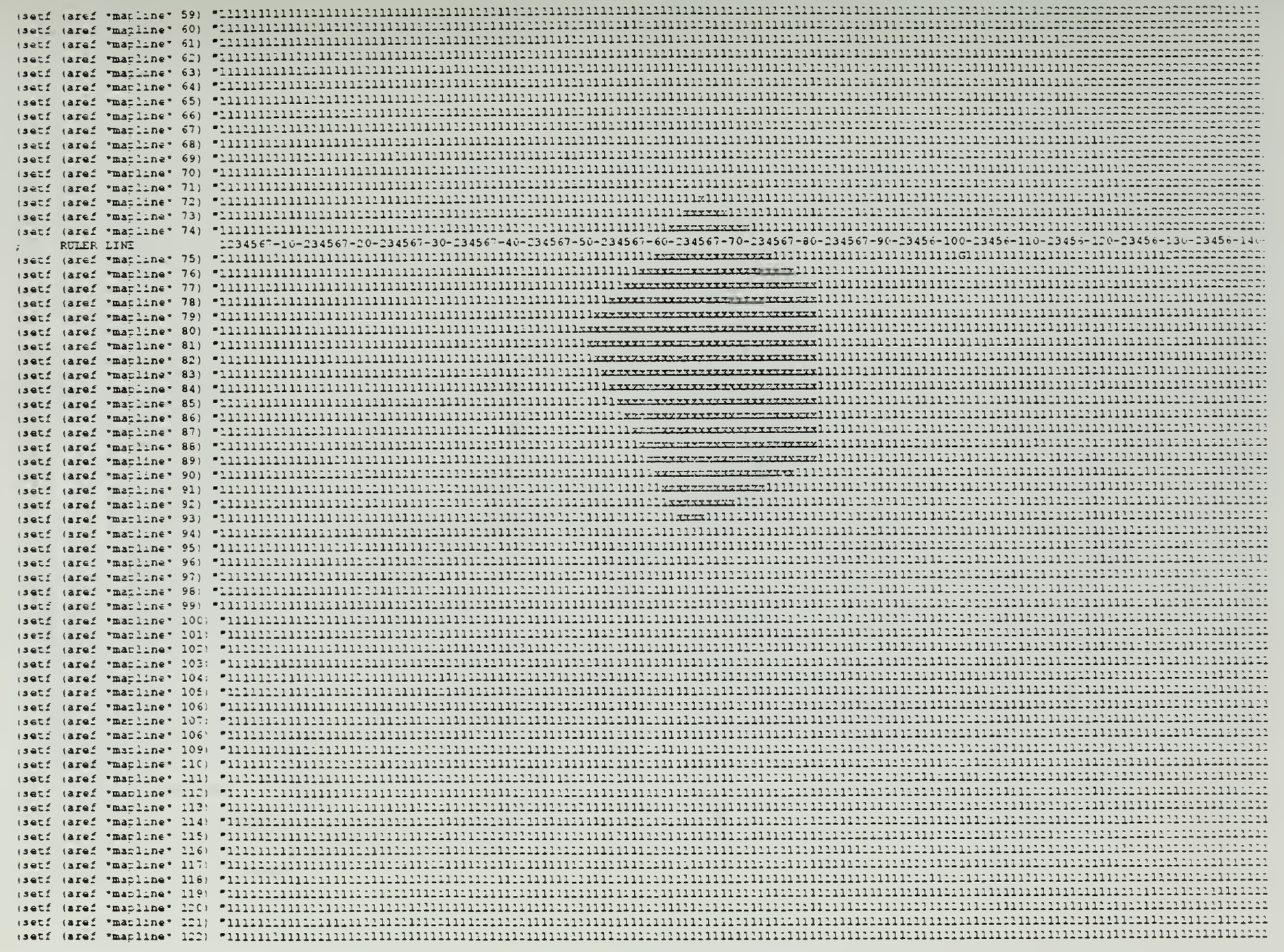

(defun draw-pt (point-coords)
  (tv:sheet-force-access (*draw-window*)
    (send *draw-window*
          :draw-point (first point-coords) (second point-coords))))
```

```
;;; -*- Mode: LISP; Syntax: Common-Lisp; Package: USER -*-

;*****************************************************************
;***** maplotlreg.lisp - contains one obstacle and one high-cost
;*****    region (c=2), with edge- and vertex-lists.

(setf *mapline* (make-array (list *map-length*)))

(setf (aref *mapline* 1)  "1111111111
```


(setf (aref *mapline* 59) "
(setf (aref *mapline* 60) "
(setf (aref *mapline* 61) "
(setf (aref *mapline* 62) "
(setf (aref *mapline* 63) "
(setf (aref *mapline* 64) "
(setf (aref *mapline* 65) "
(setf (aref *mapline* 66) "
(setf (aref *mapline* 67) "
(setf (aref *mapline* 68) "
(setf (aref *mapline* 69) "
(setf (aref *mapline* 70) "
(setf (aref *mapline* 71) "
(setf (aref *mapline* 72) "
(setf (aref *mapline* 73) "
(setf (aref *mapline* 74) "
; RULER LINE 1234567-10-234567-20-234567-30-234567-40-234567-50-234567-60-234567-70-234567-80-234567-90-23456-100-23456-110-23456-120-23456-130-23456-140
(setf (aref *mapline* 75) "
(setf (aref *mapline* 76) "
(setf (aref *mapline* 77) "
(setf (aref *mapline* 78) "
(setf (aref *mapline* 79) "
(setf (aref *mapline* 80) "
(setf (aref *mapline* 81) "
(setf (aref *mapline* 82) "
(setf (aref *mapline* 83) "
(setf (aref *mapline* 84) "
(setf (aref *mapline* 85) "
(setf (aref *mapline* 86) "
(setf (aref *mapline* 87) "
(setf (aref *mapline* 88) "
(setf (aref *mapline* 89) "
(setf (aref *mapline* 90) "
(setf (aref *mapline* 91) "
(setf (aref *mapline* 92) "
(setf (aref *mapline* 93) "
(setf (aref *mapline* 94) "
(setf (aref *mapline* 95) "
(setf (aref *mapline* 96) "
(setf (aref *mapline* 97) "
(setf (aref *mapline* 98) "
(setf (aref *mapline* 99) "
(setf (aref *mapline* 100) "
(setf (aref *mapline* 101) "
(setf (aref *mapline* 102) "
(setf (aref *mapline* 103) "
(setf (aref *mapline* 104) "
(setf (aref *mapline* 105) "
(setf (aref *mapline* 106) "
(setf (aref *mapline* 107) "
(setf (aref *mapline* 108) "
(setf (aref *mapline* 109) "
(setf (aref *mapline* 110) "
(setf (aref *mapline* 111) "
(setf (aref *mapline* 112) "
(setf (aref *mapline* 113) "
(setf (aref *mapline* 114) "
(setf (aref *mapline* 115) "
(setf (aref *mapline* 116) "
(setf (aref *mapline* 117) "
(setf (aref *mapline* 118) "
(setf (aref *mapline* 119) "
(setf (aref *mapline* 120) "
(setf (aref *mapline* 121) "
(setf (aref *mapline* 122) "

```
(setf (aref *mapline* 125) "11111111111
```

```
(154 66 9) (153 66 9  (152 66 9) (152 65 9) (151 65 9) (150 65 9) (149 65 9)
(149 64 9) (148 64 9) (147 64 9) (146 64 9) (146 63 9) (145 63 9) (144 63 9)
(143 63 9) (143 62 9) (142 62 9) (141 62 9, (140 62 9) (140 61 9) (139 61 9)
(138 61 9) (137 61 9) (137 60 9) (136 60 9) (135 60 9) (134 60 9) (134 59 9)
(133 59 9, (132 59 9) (131 59 9)

(131 58 10) (131 58 10) (129 58 10) (128 58 10) (127 58 10) (126 58 10)
(125 58 10) (124 58 10) (123 58 10) (122 58 10) (121 58 10) (120 58 10)
(119 58 10) (118 58 10) (117 58 10) (116 58 10) (115 58 10) (114 58 10)
(113 56 10) (112 58 10) (111 58 10)

(112 58 11)  (112 59 11) (113 59 11) (113 60 11) (114 60 11)
(114 61 11) (115 61 11) (115 62 11) (116 62 11) (116 63 11) (117 63 11)
(117 64 11) (118 64 11) (118 65 11) (119 65 11, (119 66 11) (120 66 11)
(120 67 11) (121 67 11) (121 68 11) (122 68 11) (122 69 11) (123 69 11)
(123 70 11) (124 70 11) (124 71 11) (125 71 11) (125 72 11) (126 72 11)
(126 73 11) (127 73 11) (127 74 11) (128 74 11)

(128 75 12) (129 75 12) (130 75 12) (131 75 12) (131 76 12) (132 76 12)
(133 76 12) (134 76 12) (134 77 12) (135 77 12) (136 77 12) (137 77 12)))

;  RULER LINE              1234567-10-234567-20-234567-30-234567-40-234567-50-234567-60-234567-70-234567-80-234567-90-23456-100-23456-110-23456-120-23456-130-23456-140-
```

APPENDIX D - HIGH-COST EXTERIOR-GOAL HCA INTERIOR-BOUNDARY CONSTRUCTION SOURCE CODE

```
/*****************************************************************************
 *****************************************************************************
 *
 *  File "bdrygen".
 *     Updated 12 Jan 89.
 *
 *  This program generates boundaries for HCA interiors
 *    and writes them to two files; "bdry_out" is a file of prolog
 *    facts recording the boundary and terrain information;
 *    "bdry_fig" is the same info ready for plotting by the "figure" utility.
 *  Requires "bgmapdata", "bgutils", "bgplotter", and "bdryjoin"
 *    to be in the same directory when started.
 *  Usage:  from unix, type:
 *              prolog
 *              [bdrygen].
 *              bg.
 *              halt.
 *
 *****************************************************************************
 *****************************************************************************/

bg :-   assert(write_flag(write)),   /* or (write_flag(no_write)) */
        initialize_bg,
        generate_bdrys,
        save(bgstate),
        tell(user),nl,write('Boundary generation done (First Pass)'),nl,nl,
        reconsult(bdryjoin),
        bdry_join.

bg2 :-  generate_bdrys,
        tell(user),write('Boundary generation done (Second Pass)'),nl,
        bdry_join.

generate_bdrys :-
        goal_point(Xg,Yg),
        region_vertices([X1,Y1,X2,Y2|RList1]),
        cons([X1,Y1,X2,Y2|RList1],[X1,Y1],RList2),
        initial_output([Xg,Yg],RList2,_),
        classify_edges(RList2,[Xg,Yg],RList3),
        convert_vlist_to_elist(1,RList3,RList4),
        assert(region_elist(RList4)),
        generate_boundaries0(RList4,[Xg,Yg]),
        order_initbdry_indices, !.

/*
 *  Temporarily, file "bgmapdata" must have a predicate for each
 *  vertex with its optimal path list.  Eventually, this should be
 *  replaced by a call to a path-finding program such as "sls" or "rri"
 */

        /*Compare 1st edge w/ 2d,3d.., recurse to comp 2d edge w/ 3d,4th..,etc*/
generate_boundaries0([X1,Y1,V12,N12,X2,Y2],[Xg,Yg]).
generate_boundaries0([X1,Y1,V12,N12,X2,Y2|Rest],[Xg,Yg]) :-
        generate_boundaries([X1,Y1,V12,N12,X2,Y2|Rest],[Xg,Yg]),
        generate_boundaries0(Rest,[Xg,Yg]).

        /* Stopping condition: only one edge left */
generate_boundaries([_,_,_,_,_,_,_],[_,_]) :- !.
        /* If vertex list from file "bgmapdata" already includes the */
        /* first point as the last, it will appear twice, so ignore  */
        /* the second occurence. */
generate_boundaries([Xa,Ya,_,_,Xb,Yb,Xa,Ya,_,_,Xb,Yb],[_,_]) :- !.
```

```
        /* Type 1 */
generate_boundaries([Xa,Ya,v,Nab,Xb,Yb,Xc,Yc,v,Ncd,Xd,Yd|Rest],[Xg,Yg]) :-
        interior_cost(Ci),exterior_cost(Ce),              /* 2vis (Type1) bdry */
        plot_2vis_bdry(Ci,Ce,Xa,Ya,Nab,Xb,Yb,Xc,Yc,Ncd,Xd,Yd), /*plot Edge1,E2*/
        generate_boundaries([Xa,Ya,v,Nab,Xb,Yb|Rest],[Xg,Yg]).  /*[E1,E3|Rest]*/

        /* Type 2 */
generate_boundaries([Xa,Ya,v,Nab,Xb,Yb,Xc,Yc,h(F),Ncd,Xd,Yd|Rest],[Xg,Yg]) :-
        interior_cost(Ci),exterior_cost(Ce),                 /* 1vis (Type 2) */
        plot_1vis_bdry(F,Ci,Ce,Xa,Ya,Nab,Xb,Yb,Xc,Yc,Ncd,Xd,Yd), /*plot E1,E2*/
        generate_boundaries([Xa,Ya,v,Nab,Xb,Yb|Rest],[Xg,Yg]). /*[E1,E3|Rest]*/

        /* Types 3 and 4 */
generate_boundaries([Xa,Ya,h(F),Nab,Xb,Yb,Xc,Yc,h(G),Ncd,Xd,Yd|Rest],[Xg,Yg]) :-
        interior_cost(Ci),exterior_cost(Ce),             /* 0vis (Type 3 or 4) */
        plot_0vis_bdry(F,G,Ci,Ce,Xa,Ya,Nab,Xb,Yb,Xc,Yc,Ncd,Xd,Yd), /*plot E1,2*/
        generate_boundaries([Xa,Ya,h(F),Nab,Xb,Yb|Rest],[Xg,Yg]).  /*[E1,E3|R]*/

initialize_bg :-
        consult(bgmapdata),
        consult(bgutils),
        consult(bgplotter),
        tell(user),nl,nl,nl,
        write('Boundaries being computed:'),nl,!.

initial_output([Xg,Yg],Region,Region_elist) :-
        assertz(title('')),
        title(Title),
        write_to_bdry_file(title,Title),
        write_to_bdry_file(goal,[Xg,Yg]),
        write_to_bdry_file(region,Region),
        /* write_to_bdry_file(region_elist,Region_elist),
        write_to_fig_file(title),
        write_to_fig_file(goal,[Xg,Yg]),
        write_heading(region),
        write_to_fig_file(region,Region), */
        !.

/* convert list of vertices (vertices are not repeated) to a list
 * of edges (a vertex appears once for each edge)
 * Also number the edges sequentially, and assert the number of edges.
 */
convert_vlist_to_elist(N,[X1,Y1,V1,X2,Y2],[X1,Y1,V1,N,X2,Y2]) :-
        assert(number_of_edges(N)).
convert_vlist_to_elist(N,[X1,Y1,o,X2,Y2|RListRest],
        [X1,Y1,o,o,X2,Y2|RevRListRest]) :-
        convert_vlist_to_elist(N,[X2,Y2|RListRest],RevRListRest), !.
convert_vlist_to_elist(N,[X1,Y1,V1,X2,Y2|RListRest],
        [X1,Y1,V1,N,X2,Y2|RevRListRest]) :-
        Nplus1 is N + 1,
        convert_vlist_to_elist(Nplus1,[X2,Y2|RListRest],RevRListRest), !.

classify_edges(L1,[Xg,Yg],L6) :-
        edge_visibility_check(L1,[Xg,Yg],L2),
        rotate_edge_list(L2,L3),
        mark_edges(L3,L4),
        insert_opposite_pt(L4,L5),
        remark_edges(L5,L6), !.

/* First step past vis edges, leaving their markings unchanged */
remark_edges([X1,Y1,v,X2,Y2|Rest],[X1,Y1,v,X2,Y2|Rest2]) :-
        remark_edges([X2,Y2|Rest],[X2,Y2|Rest2]).
/* Now step past h(b) edges. */
remark_edges([X1,Y1,h(b),X2,Y2|Rest],[X1,Y1,h(b),X2,Y2|Rest2]) :-
```

```
        remark_edges([X2,Y2|Rest],[X2,Y2|Rest2]).
/* At each h(ob) edge, change 'ob' to 'b' */
remark_edges([X1,Y1,h(ob),X2,Y2|Rest],[X1,Y1,h(b),X2,Y2|Rest2]) :-
        remark_edges([X2,Y2|Rest],[X2,Y2|Rest2]).
/* At each h(oa) edge, change 'oa' to 'a' */
remark_edges([X1,Y1,h(oa),X2,Y2|Rest],[X1,Y1,h(a),X2,Y2|Rest2]) :-
        remark_edges([X2,Y2|Rest],[X2,Y2|Rest2]).
/* At first h(a) edge, stop. */
remark_edges([X1,Y1,h(a),X2,Y2|Rest],[X1,Y1,h(a),X2,Y2|Rest]).
/* If no  h(a) edges, stop. */
remark_edges([X,Y],[X,Y]).

insert_opposite_pt(L4,L5) :-          /* On first pass, assume no shortcutting */
        not(first_pass_done),              /* occurs and set up opposite point */
        insert_tentative_opp_pt(L4,L5),    /* and optimal paths accordingly.   */
        assert_pseudo_ops(L5), !.
insert_opposite_pt(L4,L5) :-          /* On second pass, use correct opp point */
        first_pass_done,
        insert_correct_opp_pt(L4,L5), !.

/* On FIRST PASS, insert tentative opposite point at the midpoint of */
/* the opposite edges, disregarding any possible shortcutting. */
/* Change marking on other 'o' edges accordingly. */
/* First step past vis edges, leaving their markings unchanged */
insert_tentative_opp_pt([X1,Y1,v,X2,Y2|Rest],[X1,Y1,v,X2,Y2|Rest2]) :-
        insert_tentative_opp_pt([X2,Y2|Rest],[X2,Y2|Rest2]).
/* Now step past h(b) edges. */
insert_tentative_opp_pt([X1,Y1,h(b),X2,Y2|Rest],[X1,Y1,h(b),X2,Y2|Rest2]) :-
        insert_tentative_opp_pt([X2,Y2|Rest],[X2,Y2|Rest2]).
/* At first h(o) edge, branch to insert..2, passing opp edge info along. */
insert_tentative_opp_pt([X1,Y1,h(o),X2,Y2|Rest],RevisedL) :-
        optimal_path([X1,Y1|OP1],Cccw),
        insert_tentative_opp_pt2([X1,Y1,h(o),X2,Y2|Rest],[X1,Y1],Cccw,RevisedL).
/* If there is no h(a) edge, go to insert..3, then stop at last h(o) edge */
/* opposite point in previous 'o' edges. */
insert_tentative_opp_pt2([X1,Y1,h(o),X2,Y2],OE,Cccw,R2) :-
        insert_tentative_opp_pt3(X2,Y2,[X2,Y2,h(o)|OE],Cccw,R2).
/* At each h(o) edge, pass opp edge info along. */
insert_tentative_opp_pt2([X1,Y1,h(o),X2,Y2|R],OE,Cccw,R2) :-
        insert_tentative_opp_pt2([X2,Y2|R],[X2,Y2,h(o)|OE],Cccw,R2).
/* At first h(a) edge, insert 1st-guess opposite pt in previous 'o' edges. */
insert_tentative_opp_pt2([X1,Y1,h(a),X2,Y2|R],OE,Cccw,OandAList) :-
        optimal_path([X1,Y1|OPcw],Ccw),
        reverse_edge_list(OE,OERev),
        edge_length(OERev,Length),
        D is (Length+Ccw-Cccw)/2,    /* opp pt is D along the OEs ccw from pt1 */
        insert_tent_opp_pt_along_edges(OERev,D,OERevised),
        cons(OERevised,[h(a),X2,Y2|R],OandAList).
/* If there are no h(a) edges, insert 1st-guess opp pt in previous 'o' edges. */
insert_tentative_opp_pt3(X1,Y1,OE,Cccw,OandAList) :-
        optimal_path([X1,Y1|OPcw],Ccw),
        reverse_edge_list(OE,OERev),
        edge_length(OERev,Length),
        D is (Length+Ccw-Cccw)/2,    /* opp pt is D along the OEs ccw from pt1 */
        insert_tent_opp_pt_along_edges(OERev,D,OandAList).

insert_tent_opp_pt_along_edges([X1,Y1,h(o),X2,Y2|OE],D,[X1,Y1,h(ob)|OERev]) :-
        distance(X1,Y1,X2,Y2,D1), D2 is D - D1,
        D2 > 0,
        insert_tent_opp_pt_along_edges([X2,Y2|OE],D2,OERev).
insert_tent_opp_pt_along_edges([X1,Y1,h(o),X2,Y2|OE],D,
        [X1,Y1,h(ob),Xopp,Yopp,h(oa)|OERev]) :-
        distance(X1,Y1,X2,Y2,D1), D2 is D - D1,
        D2 =< 0,
```

```
        DelX is X2 - X1, DelY is Y2 - Y1,
        Xopp is X1 + DelX*(D/D1), Yopp is Y1 + DelY*(D/D1),
        assert(opposite_point(Xopp,Yopp)),
        change_o_to_oa([X2,Y2|OE],OERev).


change_o_to_oa([X,Y],[X,Y]) :- !.
change_o_to_oa([X1,Y1,h(a),X2,Y2|OE],[X1,Y1,h(a),X2,Y2|OE]) :- !.
change_o_to_oa([X1,Y1,h(o),X2,Y2|OE],[X1,Y1,h(oa),X2,Y2|OERev]) :-
        change_o_to_oa([X2,Y2|OE],[X2,Y2|OERev]), !.


/* At each vertex along tentative opposite-edge sequence, assert a */
/* pseudo-optimal-path as if no shortcutting occurred */
/* First step past visible edges */
assert_pseudo_ops([X1,Y1,v,X2,Y2|Rest]) :-
        assert_pseudo_ops([X2,Y2|Rest]).
/* Now step past 'before' edges */
assert_pseudo_ops([X1,Y1,h(b),X2,Y2|Rest]) :-
        assert_pseudo_ops([X2,Y2|Rest]).
/* At first 'opposite' edge */
assert_pseudo_ops([X1,Y1,h(ob),X2,Y2|Rest]) :-
        optimal_path([X1,Y1|OP1]),
        assert(pseudo_optimal_path([X1,Y1|OP1])),
        exterior_cost(Ce),
        assert(pseudo_optimal_path([X2,Y2,c(Ce),X1,Y1|OP1])),
        assert_pseudo_ops2([X2,Y2|Rest]).
/* At subsequent 'opposite' edges */
assert_pseudo_ops2([X1,Y1,h(ob),X2,Y2|Rest]) :-
        not(opposite_point(X1,Y1)),
        pseudo_optimal_path([X1,Y1|OP1]),
        exterior_cost(Ce),
        assert(pseudo_optimal_path([X2,Y2,c(Ce),X1,Y1|OP1])),
        assert_pseudo_ops2([X2,Y2|Rest]).
/* At edge with 'opposite point' */
assert_pseudo_ops2([X1,Y1,h(oa),X2,Y2|Rest]) :-
        opposite_point(X1,Y1),
        assert_pseudo_ops3([X2,Y2|Rest]),
        pseudo_optimal_path([X2,Y2|OP2]),
        exterior_cost(Ce),
        assert(pseudo_optimal_path([X1,Y1,c(Ce),X2,Y2|OP2])).
/* If there are no 'a' edges, assert clockwise OP at vertex and stop. */
assert_pseudo_ops3([X1,Y1]) :-
        optimal_path([X1,Y1|OP1]),
        assert(pseudo_optimal_path([X1,Y1|OP1])).
/* Search to end of 'o' edges, asserting clockwise OP at each cw vertex */
assert_pseudo_ops3([X1,Y1,h(oa),X2,Y2|Rest]) :-
        assert_pseudo_ops3([X2,Y2|Rest]),
        pseudo_optimal_path([X2,Y2|OP2]),
        exterior_cost(Ce),
        assert(pseudo_optimal_path([X1,Y1,c(Ce),X2,Y2|OP2])).
/* At first 'after' edge assert a cw ps-op and stop. */
assert_pseudo_ops3([X1,Y1,h(a),X2,Y2|Rest]) :-
        optimal_path([X1,Y1|OP1]),
        assert(pseudo_optimal_path([X1,Y1|OP1])).

/* On SECOND PASS, insert correct opposite point into the      */
/* the opposite edges, disregarding any possible shortcutting. */
/* Change marking on other 'o' edges accordingly. */
/* First step past vis edges, leaving their markings unchanged */
insert_correct_opp_pt([X1,Y1,v,X2,Y2|Rest],[X1,Y1,v,X2,Y2|Rest2]) :-
        insert_correct_opp_pt([X2,Y2|Rest],[X2,Y2|Rest2]).
/* Now step past h(b) edges. */
insert_correct_opp_pt([X1,Y1,h(b),X2,Y2|Rest],[X1,Y1,h(b),X2,Y2|Rest2]) :-
```

```
        insert_correct_opp_pt([X2,Y2|Rest],[X2,Y2|Rest2]).
/* At each h(o) edge, see if opp pt is on this edge.        */
/* If so, revise rest of h(o) edges and insert opp pt.      */
insert_correct_opp_pt([X1,Y1,h(o),X2,Y2|Rest],
        [X1,Y1,h(ob),Xopp,Yopp,h(oa),X2,Y2|Rest2]) :-
        opposite_point(Xopp,Yopp),
        on_line(Xopp,Yopp,X1,Y1,X2,Y2),
        change_o_to_oa([X2,Y2|Rest],[X2,Y2|Rest2]), !.
/* If not, mark edge h(ob) and look at rest of h(o) edges. */
insert_correct_opp_pt([X1,Y1,h(o),X2,Y2|Rest],[X1,Y1,h(ob),X2,Y2|Rest2]) :-
        insert_correct_opp_pt([X2,Y2|Rest],[X2,Y2|Rest2]).

/* Step thru edge-list, rotating it until all visible edges are on its  */
/* front, and all hidden edges are on its end.                          */
/* Upon finding a hidden edge before any vis edge, put it on the end,   */
/* and start again looking for vis or hidden edges.                     */
rotate_edge_list([X1,Y1,h,X2,Y2|Rest],RevisedList) :-
        cons([X2,Y2|Rest],[h,X2,Y2],L2),
        rotate_edge_list(L2,RevisedList), !.
/* Upon finding first vis edge, step thru the list keeping vis edges in */
/* order, and then keeping hidden edges in order.  If any vis edges are */
/* on the end of the list, put them on the front.   */
/* and if so, put them on the front, maintaining order.                 */
rotate_edge_list([X,Y,v|Rest],FullRevisedList) :-
        rotate_edge_list2([X,Y,v|Rest],RevisedList,FrontofList),
        cons(FrontofList,RevisedList,FullRevisedList), !.
/* Go past the front-end visible edges. */
rotate_edge_list2([X1,Y1,v,X2,Y2|Rest],[X1,Y1,v|L2],FrontofList) :-
        rotate_edge_list2([X2,Y2|Rest],L2,FrontofList), !.
/* Go past the first hidden edge after the visible edges. */
rotate_edge_list2([X1,Y1,h,X2,Y2|Rest],[X1,Y1,h|L2],FrontofList) :-
        rotate_edge_list3([X2,Y2|Rest],L2,FrontofList), !.
/* Go past the rest of the hidden edges after the visible edges. */
rotate_edge_list3([X1,Y1,h,X2,Y2|Rest],[X1,Y1,h|L2],FrontofList) :-
        rotate_edge_list3([X2,Y2|Rest],L2,FrontofList), !.
/* If visible edges are found past the hidden edges,  the rest will also */
/* be visible; send the rest back up to be  put on the front of the list */
rotate_edge_list3([X,Y,v|Rest],[X,Y],FrontofList) :-
        all_but_last_coords([X,Y,v|Rest],FrontofList), !.
/* Ending condition. */
rotate_edge_list3([X,Y],[X,Y],[]) :-  !.

/* eliminate the last coordinates and the last edge-vis flag */
all_but_last_coords([X,Y],[]).
all_but_last_coords([X,Y,V|Rest],[X,Y,V|RevisedRest]) :-
        all_but_last_coords(Rest,RevisedRest).

/* mark edges before (b), after (a), or opposite (o), based on whether */
/* they are before or after the opposite edge, or undetermined. */
mark_edges(L1,L3) :-
        mark_edges2(L1,L2), mark_edges3(L2,L3), assert_opposite_edge(L3).
/* 'mark_edges2' marks 'h(b)' or 'h(a)' based on opt paths of edge itself. */
/* Base case for 'mark_edges2'. */
mark_edges2([_,_],[_,_]).
/* First step past vis edges, leaving their markings unchanged */
mark_edges2([X1,Y1,v,X2,Y2|Rest],[X1,Y1,v,X2,Y2|Rest2]) :-
        mark_edges2([X2,Y2|Rest],[X2,Y2|Rest2]).
/* Upon finding a hidden edge, check if it is 'b' or 'a' or 'o' */
/* It is 'b' if opt path from X2,Y2 starts toward X1,Y1.     */
mark_edges2([X1,Y1,h,X2,Y2|Rest],[X1,Y1,h(b),X2,Y2|Rest2]) :-
        optimal_path([X2,Y2,c(C),Xi,Yi|_]),
        on_ray(Xi,Yi,X2,Y2,X1,Y1),
        mark_edges2([X2,Y2|Rest],[X2,Y2|Rest2]).
/* It is 'a' if opt path from X1,Y1 starts toward X2,Y2.     */
```

```
mark_edges2([X1,Y1,h,X2,Y2|Rest],[X1,Y1,h(a),X2,Y2|Rest2]) :-
        optimal_path([X1,Y1,c(C),Xi,Yi|_]),
        on_ray(Xi,Yi,X1,Y1,X2,Y2),
        mark_edges2([X2,Y2|Rest],[X2,Y2|Rest2]).
/* Otherwise it is potentially an opposite edge, so mark it with 'o' */
mark_edges2([X1,Y1,h,X2,Y2|Rest],[X1,Y1,h(o),X2,Y2|Rest2]) :-
        mark_edges2([X2,Y2|Rest],[X2,Y2|Rest2]).

/* mark_edges3 marks 'b' or 'a' based on previous marking of adjacent edges. */
/* First step past vis edges, leaving their markings unchanged */
mark_edges3([X1,Y1,v,X2,Y2|Rest],[X1,Y1,v,X2,Y2|Rest2]) :-
        mark_edges3([X2,Y2|Rest],[X2,Y2|Rest2]).
/* Now deal with hidden edges. */
mark_edges3([X1,Y1,H1,X2,Y2|Rest],[X1,Y1,H2,X2,Y2|Rest2]) :-
        mark_edges4([X1,Y1,H1,X2,Y2|Rest],[X1,Y1,H2,X2,Y2|Rest2]).
/* Base case #1. End of list. */
mark_edges4([X,Y],[X,Y]).
/* Base case #2. Unknown, or as previously marked. */
mark_edges4([X1,Y1,H,X2,Y2],[X1,Y1,H,X2,Y2]).
/* Base case #3. It is 'b' if next edge is already 'b'.    */
mark_edges4([X1,Y1,h(o),X2,Y2,h(b),X3,Y3],[X1,Y1,h(b),X2,Y2,h(b),X3,Y3]).
/* Base case #4. It is 'a' if previous edge is already 'a'.    */
mark_edges4([X1,Y1,h(a),X2,Y2,h(o),X3,Y3],[X1,Y1,h(a),X2,Y2,h(a),X3,Y3]).
/* Base case #5. It is still potentially an opp edge, as previously marked. */
mark_edges4([X1,Y1,H0,X2,Y2,H1,X3,Y3],[X1,Y1,H0,X2,Y2,H1,X3,Y3]).
/* It is 'b' if next edge is already 'b'.    */
mark_edges4([X1,Y1,h(o),X2,Y2,h(b),X3,Y3|Rest],
        [X1,Y1,h(b),X2,Y2,h(b),X3,Y3|Rest2]) :-
        mark_edges4([X2,Y2,h(b),X3,Y3|Rest],[X2,Y2,h(b),X3,Y3|Rest2]).
/* It is 'a' if previous edge is already 'a'.    */
mark_edges4([X1,Y1,h(a),X2,Y2,h(o),X3,Y3|Rest],
        [X1,Y1,h(a),X2,Y2,h(a),X3,Y3|Rest2]) :-
        mark_edges4([X2,Y2,h(a),X3,Y3|Rest],[X2,Y2,h(a),X3,Y3|Rest2]).
/* Otherwise it may be an opposite edge, so leave it as previously marked.   */
mark_edges4([X1,Y1,H0,X2,Y2,H1,X3,Y3|Rest],[X1,Y1,H0,X2,Y2,H2,X3,Y3|Rest2]) :-
        mark_edges4([X2,Y2,H1,X3,Y3|Rest],[X2,Y2,H2,X3,Y3|Rest2]).

/* First step past vis edges */
assert_opposite_edge([X1,Y1,v,X2,Y2|Rest]) :-
        assert_opposite_edge([X2,Y2|Rest]).
/* Second step past h(b) edges */
assert_opposite_edge([X1,Y1,h(b),X2,Y2|Rest]) :-
        assert_opposite_edge([X2,Y2|Rest]).
/* At first opposite edge, get rest of opp edge and then assert info */
assert_opposite_edge([X1,Y1,h(o),X2,Y2|Rest]) :-
        assert_opposite_edge2([X2,Y2|Rest],Rest2),
        assert(opposite_edge([X1,Y1,h(o)|Rest2])), !.
/* At first h(a) edge, or at end of list, stop.    */
assert_opposite_edge2([X,Y],[X,Y]).
assert_opposite_edge2([X1,Y1,h(a),X2,Y2|Rest],[X1,Y1]).
/* At each h(o) edge, get rest and send back opp edge vertices */
assert_opposite_edge2([X1,Y1,h(o),X2,Y2|Rest],[X1,Y1,h(o)|Rest2]) :-
        assert_opposite_edge2([X2,Y2|Rest],Rest2), !.

/*****************************************************************************/

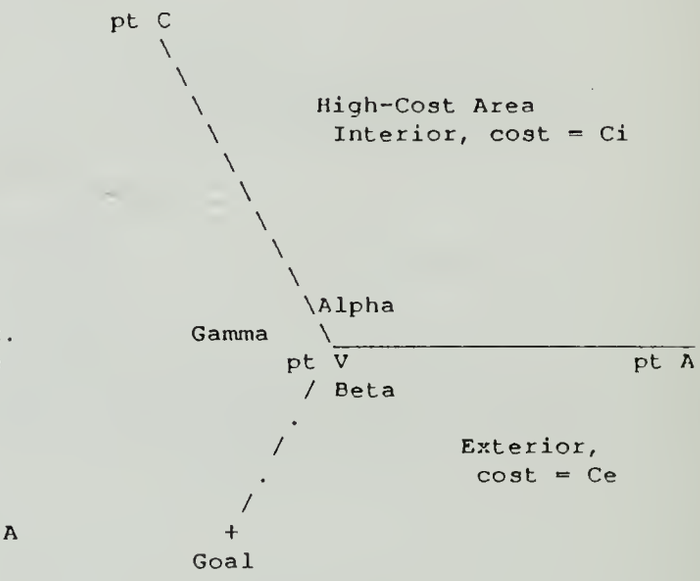
/* Succeeds if 1st pt is on a ray from 2nd pt to 3rd pt between the two pts, */
/* & fails o/w. Succeeds if 1st pt = 3rd pt, fails if 1st pt = 2nd pt. */
on_ray(X2,Y2,X1,Y1,X2,Y2).
on_ray(Xi,Yi,X1,Y1,X2,Y2) :-
        strictly_between(Xi,X1,X2),
        strictly_between(Yi,Y1,Y2),
        Yx is Xi*(Y2-Y1)/(X2-X1) + Y2 - X2*(Y2-Y1)/(X2-X1),
        within_tolerance(Xi,Yx,Xi,Yi).
```

```
/* Binds UL to the list of opposite edges. */
get_o_edges([_,_,h(v)|R],UL) :- o_edges(R,UL).
get_o_edges([_,_,h(b)|R],UL) :- o_edges(R,UL).
get_o_edges([X1,Y1,h(o),X2,Y2|R],[X1,Y1,X2,Y2|RestUL]) :-
        get_o_edges([X2,Y2|R],RestUL).
get_o_edges([_,_,h(a)|R],[]) :- get_o_edges(R,[]).
get_o_edges([_,_],[]).

order_initbdry_indices :-
        retract(initbdry([I,J],B)),
        sort([I,J],[I2,J2]),
        asserta(initbdry([I,J],B)),
        fail, !.
order_initbdry_indices :- !.
```

```
/*****************************************************************************
 *****************************************************************************
 *
 *  File "bgplotter" has the predicates which plot
 *  boundaries of various types.  It is loaded by "bg".
 *
 *****************************************************************************
 *****************************************************************************/


/*****************************************************************************
 *  This predic. plots 2vis/ (Type 1) boundaries between two HCA edges, ie,
 *  between two visible edges
 *
 *  Updated 12 Jan 89.
 *
 *
 * Ci : interior (high) cost
 * Ce : exterior (low) cost
 * Alpha : included angle of
 *         region vertex
 * Beta : angle between first
 *        edge (pt V to pt A) and
 *        a line between the
 *        vertex and the goal.
 * Gamma : angle between second
 *        edge (pt V to pt C) and
 *        a line between the
 *        vertex and the goal.
 * D1 : distance from goal to vertex.
 * RotAngle : angle needed to rotate
 *      the x-axis counterclockwise
 *      to bring it parallel with
 *      the first edge (V to A)
 * Xa,Ya : coords of first point.
 * Xv,Yv : coords of second point,
 *      the vertex, connected to pt A
 * Xc,Yc : coords of third point.
 * Xv2,Yv2 : if the edges are not
 *      connected, these are the
 *      coords of the "inner" vertex
 *      of the second edge, of which
 *      pt C is the other vertex.
 */
```




```
plot_2vis_bdry(Ci,Ce,Xa,Ya,Nab,Xb,Yb,Xc,Yc,Ncd,Xd,Yd) :-
        virtual_vertex(Xa,Ya,Xb,Yb,Xc,Yc,Xd,Yd,Xvv,Yvv),
        plot_2vis_bdry_allcases(Ci,Ce,Nab,Ncd,Xa,Ya,Xvv,Yvv,Xd,Yd), !.
plot_2vis_bdry_allcases(Ci,Ce,N1,N2,Xa,Ya,Xv,Yv,Xc,Yc) :-
        initialize_for_b1,                          /* This is a 2v/ or */
        goal_point(Xg,Yg),                          /* Type 1 boundary. */
        pi(Pi),
        distance(Xv,Yv,Xg,Yg,D1),
        distance(Xv,Yv,Xc,Yc,D2),
        distance(Xa,Ya,Xv,Yv,D3),
        distance(Xa,Ya,Xc,Yc,D4),
        distance(Xa,Ya,Xg,Yg,D5),
        not(D1=0),not(D2=0),not(D3=0),          /* If any of these fail,*/
        not(D4=0),not(D5=0),        /* => programming error or map error */
        Cos1 is (D2^2+D3^2-D4^2)/(2*D2*D3),
        arccos(Cos1,Alpha),
        Cos2 is (D1^2+D3^2-D5^2)/(2*D1*D3),
        arccos(Cos2,Beta),
        Gamma is 2*Pi - Alpha - Beta,
```

```
        ThetaCrit is asin(Ce/Ci),
        tell(user), write('2vis (Type 1) bdry being plotted between edges '),
        write(N1),write(' and '),write(N2),nl,
        check_input_for_type1(Alpha,Beta,Gamma,ThetaCrit,D1),
        compute_angle_of_rotation(Xv,Yv,Xa,Ya,RotAngle),
        assert(bdry([Xv,Yv])),
        calc_bdry_type1(Alpha,Beta,Gamma,ThetaCrit,D1,Xa,Ya,Xv,Yv,Xc,Yc),
        rotate_bdry(RotAngle),
        bdry(Bdry),
        reverse_path_list(Bdry,RevBdry),
        truncate_off_map(RevBdry,FinalBdry),
        assert(initbdry([N1,N2],FinalBdry)),
        /* tell(bdry_out), write(' Type 1 '),nl,
        write_to_bdry_file(bdry,Bdry,N1,N2),nl,
        output_to_figure_file, */
        !.
plot_2vis_bdry_allcases(_,_,_,_,_,_,_,_,_) :-  !.

initialize_for_b1 :-
        abolish(done,0),
        abolish(bdry,1),
        abolish(theta1,1), !.

calc_bdry_type1(A,B,G,Tc,D1,Xa,Ya,Xv,Yv,Xc,Yc) :-
        pi(Pi),
        get_T1_range(G,Xv,Yv,Xc,Yc,T1min,T1max),
        precision(Precision),
        DelT1 is (T1max-T1min)/(Precision/2),
        T1init is T1min - DelT1/2,    /* let 1st point be closer to vertex */
        assert(theta1(T1init)),
        retract(theta1(T1prev)),
        T1 is T1prev + DelT1,
        T1 < T1max+0.01,
        T1 < Pi/2,
        assert(theta1(T1)),
        calc_bdry1_pt(D1,T1,A,B,G,Tc,T3,T4,Y1,Y2),
        store_2vis_results(-T3,-T4,Y1,Y2,B,D1,Xv,Yv),
        done, !.
calc_bdry_type1(_,_,_,_,_,_,_,_,_,_,_) :- !.


calc_bdry1_pt(D1,T1,A,B,G,Tc,
                T3,T4,Y1,Y2) :-
        abolish(done1,0),
        abolish(theta3,1),
        abolish(increment,1),
        pi(Pi),
        InitIncr is -(Pi-B)/2,          /* 1/2 of range of Theta3 */
        assert(increment(InitIncr)),
        T3init is B-(Pi/2),
        assert(theta3(T3init)),
        retract(theta3(T3)),
        calc_Epsilon(D1,T1,A,B,G,Tc,T3,T4,Y1,Y2,E),
        get_direction(E,Direction),
        get_T3new(Direction,T3,T3new,Incr),
        assert(theta3(T3new)),
        done1, !.

virtual_vertex(X1,Y1,X2,Y2,X3,Y3,X4,Y4,Xv,Yv) :-    /* the virtual vertex is */
        line_intersection(X1,Y1,X2,Y2,X3,Y3,X4,Y4,Xv,Yv),!.  /* the point of */
                                                /* intersection of the lines. */

get_T1_range(G,Xv,Yv,Xc,Yc,T1init,T1final) :-
        pi(Pi),
```

```
        T1init is G-Pi/2,
        goal_point(Xg,Yg),
        distance(Xg,Yg,Xv,Yv,D1),
        distance(Xc,Yc,Xv,Yv,D2),
        distance(Xg,Yg,Xc,Yc,D3),
        Cos is (D2^2+D3^2-D1^2)/(2*D2*D3),
        arccos(Cos,Angle),
        get_T1final(Angle,T1final), !.

get_T1final(Angle,T1final) :-
        Angle > 0.05,
        pi(Pi),
        T1final is Pi/2 - Angle, !.
get_T1final(Angle,T1final) :-                /* Make Thetal a little larger */
        Angle =< 0.05,
        pi(Pi),
        T1final is ((Pi/2 - Angle)+(Pi/2))/2, !.

get_direction(E,minus) :-
        E > 0.001,!.
get_direction(E,plus) :-
        E < -0.001,!.
get_direction(E,done) :-
        assert(done1), !.

get_T3new(done,_,_,_) :- !.
get_T3new(plus,T3,T3new,Incr) :-
        increment(Incr),
        Incr < 0,                            /* If direction of search */
        T3new is T3-Incr,                    /* has changed, halve the */
        HalfIncr is -Incr/2,                 /* incr & change signs,   */
        retract(increment(_)),               /* otherwise don't.       */
        assert(increment(HalfIncr)), !.
get_T3new(plus,T3,T3new,Incr) :-
        retract(increment(Incr)),
        T3new is T3+Incr,
        assert(increment(Incr)), !.
get_T3new(minus,T3,T3new,Incr) :-
        increment(Incr),                     /* If direction of search */
        Incr > 0,                            /* has changed, halve the */
        T3new is T3-Incr,                    /* incr & change signs,   */
        HalfIncr is -Incr/2,                 /* otherwise don't.       */
        retract(increment(_)),
        assert(increment(HalfIncr)), !.
get_T3new(minus,T3,T3new,Incr) :-
        retract(increment(Incr)),
        T3new is T3+Incr,
        assert(increment(Incr)), !.

calc_Epsilon(DI,T1,A,B,G,Tc,T3,T4,Y1,Y2,E) :-
        T2 is asin(sin(Tc)*sin(T1)),
        T4 is asin(sin(Tc)*sin(T3)),
        SinTc is sin(Tc),
        X1 is D1*sin(G)/cos(T1),
        Y1 is D1*sin(B)/cos(T3),
        A1 is D1*sin(A)/cos(T2),
        A2 is D1*sin(A)/cos(T4),
        B1 is cos(T1-G)/cos(T1),
        B2 is cos(T2+A)/cos(T2),
        B3 is cos(T3-B)/cos(T3),
        B4 is cos(T4+A)/cos(T4),
        X2 is A1*(B3-B1*B4)/(1-B2*B4),
        Y2 is A2*B1-(X2*cos(T2+A)/cos(T4)),
        Lhs is SinTc*X1 + X2,
```

```
        Rhs is SinTc*Y1 + Y2,
        E is Lhs-Rhs, !.        /* E > 0  if  Cost(X-path) > Cost(Y-path) */

check_input_for_type1(A,B,G,Tc,D1) :-
        pi(Pi),
        A > 0, A < Pi,
        B > Pi-A, B < Pi,
        G > Pi-A, G < Pi,
        Tc > 0,
        Tc < Pi/2,
        D1 > 0, !.
check_input_for_type1(Alpha,Beta,Gamma,Thetacrit,D1) :-
        convert_rads_to_degr(Alpha,AlphaDeg),
        convert_rads_to_degr(Beta,BetaDeg),
        convert_rads_to_degr(Gamma,GammaDeg),
        convert_rads_to_degr(Thetacrit,ThetacritDeg),
        tell(user),
        write('           ERROR in type 1 input:  A='),
        write(AlphaDeg),
        write('  B='),
        write(BetaDeg),
        write('  G='),
        write(GammaDeg),
        write('  D1='),
        write(ThetacritDeg),
        write(D1),nl,
        fail, !.

/*****************************************************************************
 *
 * This predic. draws 1vis (Type 2) boundaries, ie, boundaries between
 * one visible and one hidden HCA edge.
 *
 * Updated 31 Jan 89
 *
 *  "plot_1vis_bdry" draws a boundary for edge 1 visible and edge 2 hidden;
 *  "plotbdry2_inv" draws a boundary for edge 2 visible and edge 1 hidden;
 *
 *                                        pt D
 * Ci : interior (high) cost                 /
 * Ce : exterior (low) cost                 /   High-Cost Area
 * Alpha : included angle between     pt P1 /  Interior, cost = Ci
 *         the two edges.                  /:
 * Beta : angle between first             / :
 *        edge (pt V to pt A) and        /  :   Alpha
 *        a line between the          pt C \ :
 *        vertex and the goal.             \:Gamma
 * X1 : distance from goal to vertex.       :_____________________
 * Angle : angle needed to rotate       pt B \                  pt A
 *      the x-axis counterclockwise            .  Beta
 *      to bring it parallel with               \
 *      the first edge (V to A)                  .          Exterior,
 * Xv,Yv : coords of second point,               \          cost = Ce
 *      the vertex.                               +
 *                                               Goal
 *
 *                                  Here the rotation angle = 0
 *
 */

plot_1vis_bdry(o,Ci,Ce,Xa,Ya,Nab,Xb,Yb,Xc,Yc,Ncd,Xd,Yd) :-  /* Opposite edge. */
        not(opposite_point(Xc,Yc)),
        pseudo_optimal_path([Xc,Yc|OPc]),        /* 1st-pass, no shortcutting */
        counterclockwise([Xc,Yc|OPc]),          /* If before opposite point.*/
```

```
        tell(user),
        write('lv/b (Type 2) bdry being plotted between edges '),
        write(Nab),write(' and '),write(Ncd),nl,
        goal_point(Xg,Yg),                              /* lv/- (Type 2) bdry */
        translate_line(Xb,Yb,Xc,Yc,Xd,Yd,Xdtr,Ydtr),
        path_length([Xc,Yc|OPc],D1),
        distance(Xb,Yb,Xc,Yc,D2calc),
        add_epsilon_if_zero(D2calc,D2),
        distance(Xb,Yb,Xg,Yg,D3),
        distance(Xc,Yc,Xd,Yd,Dcd),
        distance(Xa,Ya,Xb,Yb,Dab),
        distance(Xa,Ya,Xg,Yg,Dag),
        distance(Xa,Ya,Xdtr,Ydtr,Dadtr),
        distance(Xa,Ya,Xc,Yc,Dca),
        Cos1 is (Dcd^2+Dab^2-Dadtr^2)/(2*Dcd*Dab),
        arccos(Cos1,AlphaAbs),
        sign_of_Alpha(AlphaAbs,Xa,Ya,Xb,Yb,Xc,Yc,Xd,Yd,Alpha),
        Cos2 is (D3^2+Dab^2-Dag^2)/(2*D3*Dab),
        arccos(Cos2,Beta),
        Cos3 is (D2^2+Dab^2-Dca^2)/(2*D2*Dab),
        arccos(Cos3,Gamma),
        compute_angle_of_rotation(Xd,Yd,Xc,Yc,RotA),
        connected(Xa,Ya,Xb,Yb,Xc,Yc,Xd,Yd,Conn),
        plot_lvis_bdry2(before,Conn,Ci,Ce,Nab,Ncd,
                Alpha,Beta,Gamma,D1,D2,D3,
                Dag,Dab,Xc,Yc,RotA), !.

plot_lvis_bdry(o,Ci,Ce,Xa,Ya,Nab,Xb,Yb,Xc,Yc,Ncd,Xd,Yd) :-   /* opposite edge */
        pseudo_optimal_path([Xd,Yd|OPd]),             /* Must be after opp pt */
        tell(user),
        write('lv/a (Type 2) bdry being plotted between edges '),
        write(Nab),write(' and '),write(Ncd),nl,
        goal_point(Xg,Yg),                              /* lv/- (Type 2) bdry */
        translate_line(Xa,Ya,Xd,Yd,Xc,Yc,Xctr,Yctr),
        path_length([Xd,Yd|OPd],D1),
        distance(Xa,Ya,Xd,Yd,D2calc),
        add_epsilon_if_zero(D2calc,D2),
        distance(Xa,Ya,Xg,Yg,D3),
        distance(Xc,Yc,Xd,Yd,Dcd),
        distance(Xa,Ya,Xb,Yb,Dab),
        distance(Xb,Yb,Xg,Yg,Dbg),
        distance(Xb,Yb,Xctr,Yctr,Dbctr),
        distance(Xb,Yb,Xd,Yd,Ddb),
        Cos1 is (Dcd^2+Dab^2-Dbctr^2)/(2*Dcd*Dab),
        arccos(Cos1,AlphaAbs),
        sign_of_Alpha(AlphaAbs,Xb,Yb,Xa,Ya,Xd,Yd,Xc,Yc,Alpha),
        Cos2 is (D3^2+Dab^2-Dbg^2)/(2*D3*Dab),
        arccos(Cos2,Beta),
        Cos3 is (D2^2+Dab^2-Ddb^2)/(2*D2*Dab),
        arccos(Cos3,Gamma),
        compute_angle_of_rotation(Xd,Yd,Xc,Yc,RotA),
        connected(Xa,Ya,Xb,Yb,Xc,Yc,Xd,Yd,Conn),
        plot_lvis_bdry2(after,Conn,Ci,Ce,Nab,Ncd,Alpha,Beta,Gamma,
                        D1,D2,D3,Dbg,Dab,Xd,Yd,RotA),!.

plot_lvis_bdry(b,Ci,Ce,Xa,Ya,Nab,Xb,Yb,Xc,Yc,Ncd,Xd,Yd) :- /* before opp edge */
        optimal_path([Xd,Yd,c(C),Xp1,Yp1|F]),
        counterclockwise([Xd,Yd,c(C),Xp1,Yp1|F]),
        tell(user), write('lv/b (Type 2) bdry being plotted between edges '),
        write(Nab),write(' and '),write(Ncd),nl,
        goal_point(Xg,Yg),                              /* lv/- (Type 2) bdry */
        translate_line(Xb,Yb,Xc,Yc,Xd,Yd,Xdtr,Ydtr),
        distance(Xd,Yd,Xp1,Yp1,Ddp),
        assert_shortcut_flag(Xc,Yc,Xp1,Yp1),
```

```
        path_length([Xp1,Yp1|P],D1),
        distance(Xb,Yb,Xp1,Yp1,D2calc),
        add_epsilon_if_zero(D2calc,D2),
        distance(Xb,Yb,Xg,Yg,D3),
        distance(Xc,Yc,Xd,Yd,Dcd),
        distance(Xa,Ya,Xb,Yb,Dab),
        distance(Xa,Ya,Xg,Yg,Dag),
        distance(Xa,Ya,Xdtr,Ydtr,Dadtr),
        distance(Xa,Ya,Xp1,Yp1,Dp1a),
        Cos1 is (Dcd^2+Dab^2-Dadtr^2)/(2*Dcd*Dab),
        arccos(Cos1,AlphaAbs),
        sign_of_Alpha(AlphaAbs,Xa,Ya,Xb,Yb,Xc,Yc,Xd,Yd,Alpha),
        Cos2 is (D3^2+Dab^2-Dag^2)/(2*D3*Dab),
        arccos(Cos2,Beta),
        Cos3 is (D2^2+Dab^2-Dp1a^2)/(2*D2*Dab),
        arccos(Cos3,Gamma),
        compute_angle_of_rotation(Xd,Yd,Xc,Yc,RotA),
        connected(Xa,Ya,Xb,Yb,Xc,Yc,Xd,Yd,Conn),
                        /* changed Xc,Yc for Xp1,Yp1 */
        plot_lvis_bdry2(before,Conn,Ci,Ce,Nab,Ncd,Alpha,Beta,Gamma,
                        D1,D2,D3,Dag,Dab,Xp1,Yp1,RotA),!.

plot_lvis_bdry(a,Ci,Ce,Xa,Ya,Nab,Xb,Yb,Xc,Yc,Ncd,Xd,Yd) :-  /* after opp edge */
        optimal_path([Xc,Yc,c(C),Xp1,Yp1|P]),
        clockwise([Xc,Yc,c(C),Xp1,Yp1|P]),          /* discriminates btwn OP's */
                                        /* in opposite directions from opp pt. */
        tell(user), write('lv/a (Type 2) bdry being plotted between edges '),
        write(Nab),write(' and '),write(Ncd),nl,
        goal_point(Xg,Yg),                              /* lv/- (Type 2) bdry */
        translate_line(Xa,Ya,Xd,Yd,Xc,Yc,Xctr,Yctr),
        distance(Xc,Yc,Xp1,Yp1,Dcp),
        assert_shortcut_flag(Xd,Yd,Xp1,Yp1),
        path_length([Xp1,Yp1|P],D1),
        distance(Xa,Ya,Xp1,Yp1,D2calc),
        add_epsilon_if_zero(D2calc,D2),
        distance(Xa,Ya,Xg,Yg,D3),
        distance(Xc,Yc,Xd,Yd,Dcd),
        distance(Xa,Ya,Xb,Yb,Dab),
        distance(Xb,Yb,Xg,Yg,Dbg),
        distance(Xb,Yb,Xctr,Yctr,Dbctr),
        distance(Xb,Yb,Xp1,Yp1,Dp1b),
        Cos1 is (Dcd^2+Dab^2-Dbctr^2)/(2*Dcd*Dab),
        arccos(Cos1,AlphaAbs),
        sign_of_Alpha(AlphaAbs,Xb,Yb,Xa,Ya,Xd,Yd,Xc,Yc,Alpha),
        Cos2 is (D3^2+Dab^2-Dbg^2)/(2*D3*Dab),
        arccos(Cos2,Beta),
        Cos3 is (D2^2+Dab^2-Dp1b^2)/(2*D2*Dab),
        arccos(Cos3,Gamma),
        compute_angle_of_rotation(Xd,Yd,Xc,Yc,RotA),
        connected(Xa,Ya,Xb,Yb,Xc,Yc,Xd,Yd,Conn),
                /* changed Xd,Yd for Xp1,Yp1 (why Xd,Yd???) */
        plot_lvis_bdry2(after,Conn,Ci,Ce,Nab,Ncd,Alpha,Beta,Gamma,
                        D1,D2,D3,Dbg,Dab,Xp1,Yp1,RotA),!.
/* plot_lvis_bdry(_,_,_,_,_,_,_,_,_,_,_,_,_) :-  !. */

plot_lvis_bdry2(before,Conn,Ci,Ce,N1,N2,A,B,G,D1,D2,D3,Dag,Dab,Vx,Vy,RotAngle):-
        abolish(bdry,1),
        abolish(done,0),
        abolish(theta1,1),
        calc_lvis_bdry(Ci,Ce,A,B,G,D1,D2,D3,Dag,Dab,Vx,Vy),
        rotate_bdry(RotAngle),
        remove_last_bdry_coord_if_disconnected(Conn),
        bdry(Bdry),
        reverse_path_list(Bdry,RevBdry),
```

```
        truncate_off_map(RevBdry,FinalBdry),
        assert(initbdry([N1,N2],FinalBdry)),
/*      tell(bdry_out),write(' Type 2 '),nl,
        write_to_bdry_file(bdry,Bdry),
        output_to_figure_file, */
        !.
plot_1vis_bdry2(after,Conn,Ci,Ce,N1,N2,A,B,G,D1,D2,D3,Dbg,Dab,Vx,Vy,RotAngle):-
        abolish(bdry,1),
        abolish(done,0),
        abolish(theta1,1),
        calc_1vis_bdry(Ci,Ce,A,B,G,D1,D2,D3,Dbg,Dab,Vx,Vy),
        invert_bdry,
        rotate_bdry(RotAngle),
        remove_last_bdry_coord_if_disconnected(Conn),
        bdry(Bdry),
        reverse_path_list(Bdry,RevBdry),
        truncate_off_map(RevBdry,FinalBdry),
        assert(initbdry([N1,N2],FinalBdry)),
/*      tell(bdry_out),write(' Type 2-inv '),nl,
        write_to_bdry_file(bdry,Bdry),
        output_to_figure_file, */
        !.

calc_1vis_bdry(Ci,Ce,A,B,G,D1,D2,D3,Dag,Dab,Vx,Vy) :-
        assert(bdry([Vx,Vy])),
        maxX(XMax), minX(XMin),
        LargeNumber is (XMax - XMin)*100,
        assert(sc_bdry_pt_dist(LargeNumber)),
        CostRatio is Ce/Ci,
        Tc is asin(CostRatio),
        get_t1min(Dag,Dab,D3,T1min),
        get_t1max(B,T1max),
        precision(Precision),
        DelT1 is (T1max-T1min)/Precision,
        T1init is T1max + DelT1,
        assert(theta1(T1init)),
        retract(theta1(T1prev)),
        T1 is T1prev - DelT1,
        T1 >= T1min,
        assert(theta1(T1)),
        calc_1vis_bdry_pt(Tc,A,B,G,D1,D2,D3,T1,X1,X2),
        store_1vis_results(X1,X2,Tc,Vx,Vy), fail.
calc_1vis_bdry(_,_,_,_,_,_,_,_,_,_,_,_,_) :-
        abolish(sc_bdry_pt_dist,1), !.

get_t1min(Dag,Dab,D3,T1min) :-
        pi(Pi),
        Cos1 is (Dag^2+Dab^2-D3^2)/(2*Dag*Dab),
        arccos(Cos1,T1minplus90),
        T1min is T1minplus90 - Pi/2, !.
get_t1max(B,T1max)  :-
        pi(Pi),
        NinetyminusB is Pi/2 - B,
        T1max is NinetyminusB, !.

calc_1vis_bdry_pt(Tc,A,B,G,D1,D2,D3,T1,X1,X2) :-
        T2 is asin(sin(T1)*sin(Tc)),
        T1plusB is T1 + B,
        TcplusAminusT2 is Tc + A - T2,
        T2minusA is T2 - A,
        T2minusG is T2 - G,
        F1 is sin(A) - cos(T2)*sin(Tc),
        X2 is (-D1*sin(Tc)*cos(T2)*cos(T2minusA)
                +D2*(cos(T2minusA)*sin(G)-cos(T2minusG)*F1)
```

```
                +D3*(cos(T2)/cos(T1))*
                        (sin(B)*sin(Tc)*cos(T2minusA)+cos(T1plusB)*F1))
                /(sin(TcplusAminusT2)*F1
                        +cos(T2minusA)*(cos(Tc+A)+cos(T2))),
        X1 is -X2*(sin(TcplusAminusT2)/cos(T2minusA))
                -D2*(cos(T2minusG)/cos(T2minusA))
                +D3*(cos(T2)*cos(T1plusB)/(cos(T1)*cos(T2minusA))), !.

translate_line(Xref,Yref,X1,Y1,X2,Y2,X2trans,Y2trans) :-
        DelX is X1-Xref,
        DelY is Y1-Yref,
        X2trans is X2 - DelX,
        Y2trans is Y2 - DelY, !.

next_to_last_pt([X,Y,Xlast,Ylast],[X,Y]).
next_to_last_pt([X1,Y1|R],[X,Y]) :-
        next_to_last_pt(R,[X,Y]), !.

store_lvis_results(X1,X2,Tc,Vx,Vy) :-
        not(shortcut(_,_,Vx,Vy)),
        Xbdry is Vx - X1 - X2*sin(Tc),
        Ybdry is Vy + X2*cos(Tc),
        retract(bdry(BList)),
        assert(bdry([Xbdry,Ybdry|BList])), !.
store_lvis_results(X1,X2,Tc,Vx,Vy) :-          /* The effect of rules 2 & 3  */
        shortcut(_,_,Vx,Vy),                   /* is to exclude the initial  */
        Xbdry is Vx - X1 - X2*sin(Tc),         /* portion of a bdry which    */
        Ybdry is Vy + X2*cos(Tc),              /* starts at a s/c pt, as long*/
        distance(Xbdry,Ybdry,Vx,Vy,Dnew),      /* as the bdry is coming back */
        sc_bdry_pt_dist(Dold),                 /* toward the s/c pt, and     */
        Dnew >= Dold,                          /* include the later portion  */
        retract(sc_bdry_pt_dist(Dold)),        /* as it goes away from it,   */
        assert(sc_bdry_pt_dist(Dnew)),         /* since thetamax is calculated*/
        retract(bdry(BList)),                  /* for the non-s/c case and is*/
        assert(bdry([Xbdry,Ybdry|BList])), !. /* too large for the s/c case.*/
store_lvis_results(X1,X2,Tc,Vx,Vy) :-
        shortcut(_,_,Vx,Vy),
        Xbdry is Vx - X1 - X2*sin(Tc),
        Ybdry is Vy + X2*cos(Tc),
        distance(Xbdry,Ybdry,Vx,Vy,Dnew),
        retract(sc_bdry_pt_dist(Dold)),
        Dnew < Dold,
        assert(sc_bdry_pt_dist(Dnew)), !.


/* If edge AB is parallel to CD, AlphaAbs will be 0, so Alpha is 0. */
sign_of_Alpha(0,Xa,Ya,Xb,Yb,Xc,Yc,Xd,Yd,0).
/* If AB intersects CD on the B-side of AB, Alpha is positive. */
sign_of_Alpha(AlphaAbs,Xa,Ya,Xb,Yb,Xc,Yc,Xd,Yd,AlphaAbs) :-
        line_intersection(Xa,Ya,Xb,Yb,Xc,Yc,Xd,Yd,Xi,Yi),
        distance(Xi,Yi,Xa,Ya,Dia),
        distance(Xi,Yi,Xb,Yb,Dib),
        Dia >= Dib.
/* Otherwise, AB intersects CD on the A-side of AB and Alpha is negative. */
sign_of_Alpha(AlphaAbs,Xa,Ya,Xb,Yb,Xc,Yc,Xd,Yd,-AlphaAbs).

/* If X=0, return a slightly positive value, else leave X unchanged */
add_epsilon_if_zero(0,0.0001).
add_epsilon_if_zero(X,X).

/* returns Conn = conn if the two line segments are connected, */
/* and Conn = disc otherwise.  */
connected(Xa,Ya,Xb,Yb,Xc,Yc,Xa,Ya,conn).
connected(Xa,Ya,Xb,Yb,Xb,Yb,Xd,Yd,conn).
```

```
connected(Xa,Ya,Xb,Yb,Xa,Ya,Xd,Yd,conn).
connected(Xa,Ya,Xb,Yb,Xc,Yc,Xb,Yb,conn).
connected(Xa,Ya,Xb,Yb,Xc,Yc,Xd,Yd,disc).

/* If the two edges are not connected, the first point in the */
/* Bdry list is not part of the boundary, but only there to */
/* specify the point about which to rotate. */
remove_last_bdry_coord_if_disconnected(conn).
remove_last_bdry_coord_if_disconnected(disc) :-
        retract(bdry(Bdry)),
        reverse_path_list(Bdry,[X,Y|ReversedBdry]),
        reverse_path_list(ReversedBdry,RevisedBdry),
        assert(bdry(RevisedBdry)).


/*****************************************************************************
 *
 * Plot 0vis boundaries, between two hidden edges.
 *
 * Updated 12 Jan 89.
 */

plot_0vis_bdry(b,b,Ci,Ce,Xa,Ya,Nab,Xb,Yb,Xc,Yc,Ncd,Xd,Yd) :-
        tell(user),                          /* 'before' compared with 'before' */
        write('0v/M(b) (Type 3) bdry being plotted between edges '),
        write(Nab),write(' and '),write(Ncd),nl,
        abolish(bdry,1),
        optimal_path([Xd,Yd,c(Ca),Xp1,Yp1|OPp1]),
        counterclockwise([Xd,Yd,c(Ca),Xp1,Yp1|OPp1]),
        path_length([Xp1,Yp1|OPp1],Dp1g),
        optimal_path([Xb,Yb,c(Cb),Xp3,Yp3|OPp3]),
        path_length([Xp3,Yp3|OPp3],D1),
        D2 is Dp1g - D1,
        distance(Xa,Ya,Xb,Yb,Dab),
        distance(Xa,Ya,Xp1,Yp1,D3),
        distance(Xb,Yb,Xp1,Yp1,Dbp1),
        distance(Xd,Yd,Xp1,Yp1,Ddp1),
        distance(Xa,Ya,Xd,Yd,Dad),
        distance(Xc,Yc,Xd,Yd,Dcd),
        distance(Xp3,Yp3,Xa,Ya,Z),
        pi(Pi),
        Cos1 is (D3^2+Dab^2-Dbp1^2)/(2*D3*Dab),
        arccos(Cos1,A),
        Cos2 is (Ddp1^2+D3^2-Dad^2)/(2*Ddp1*D3),
        arccos(Cos2,PiminusBeta),
        B is Pi - PiminusBeta,
        Tc is asin(Ce/Ci),
        compute_angle_of_rotation(Xd,Yd,Xc,Yc,RotAngle),
        calc_bdry_pt_0visM(A,B,Tc,Xp1,Yp1,Z,D1,D2,D3,0,X2bdry,Y2bdry),
        calc_bdry_pt_0visM(A,B,Tc,Xp1,Yp1,Z,D1,D2,D3,Dcd,X1bdry,Y1bdry),
        abolish(bdry,1),
        assert(bdry([X1bdry,Y1bdry,X2bdry,Y2bdry,Xp1,Yp1])),
        rotate2_bdry(RotAngle),
        bdry([X1,Y1,X2,Y2]),
        correct_error_in_conn_edges(Xb,Yb,Xc,Yc,Xp1,Yp1,X2,Y2,X2r,Y2r),
        assert(initbdry([Nab,Ncd],[X2r,Y2r,X1,Y1])), !.

plot_0vis_bdry(a,a,Ci,Ce,Xa,Ya,Nab,Xb,Yb,Xc,Yc,Ncd,Xd,Yd) :-
        tell(user),                          /* 'after' compared with 'after' */
        write('0v/M(a) (Type 3) bdry being plotted between edges '),
        write(Nab),write(' and '),write(Ncd),nl,
        abolish(bdry,1),
        optimal_path([Xc,Yc,c(Cc),Xp3,Yp3|OPp3]),
        path_length([Xp3,Yp3|OPp3],D1),
```

```
        optimal_path([Xa,Ya,c(C),Xp1,Yp1|OPp1]),
        clockwise([Xa,Ya,c(C),Xp1,Yp1|OPp1]),
        path_length([Xp1,Yp1|OPp1],Dp1g),
        D2 is Dp1g - D1,
        distance(Xa,Ya,Xb,Yb,Dab),
        distance(Xd,Yd,Xp1,Yp1,D3),
        distance(Xa,Ya,Xp1,Yp1,Dap1),
        distance(Xc,Yc,Xp1,Yp1,Dcp1),
        distance(Xa,Ya,Xd,Yd,Dad),
        distance(Xa,Ya,Xb,Yb,Dab),
        distance(Xc,Yc,Xd,Yd,Dcd),
        distance(Xp3,Yp3,Xd,Yd,Z),
        pi(Pi),
        Cos1 is (D3^2+Dcd^2-Dcp1^2)/(2*D3*Dcd),
        arccos(Cos1,A),
        Cos2 is (Dap1^2+D3^2-Dad^2)/(2*Dap1*D3),
        arccos(Cos2,PiminusBeta),
        B is Pi - PiminusBeta,
        Tc is asin(Ce/Ci),
        compute_angle_of_rotation(Xb,Yb,Xa,Ya,RotAngle),
        calc_bdry_pt_0visM(A,B,Tc,Xp1,Yp1,Z,D1,D2,D3,0,X1bdry,Y1bdry),
        calc_bdry_pt_0visM(A,B,Tc,Xp1,Yp1,Z,D1,D2,D3,Dab,X2bdry,Y2bdry),
        abolish(bdry,1),
        assert(bdry([X2bdry,Y2bdry,X1bdry,Y1bdry,Xp1,Yp1])),
        invert_bdry,
        rotate2_bdry(RotAngle),
        bdry([X1,Y1,X2,Y2]),
        correct_error_in_conn_edges(Xb,Yb,Xc,Yc,Xp1,Yp1,X2,Y2,X2r,Y2r),
        assert(initbdry([Nab,Ncd],[X2r,Y2r,X1,Y1])), !.

plot_0vis_bdry(b,a,Ci,Ce,Xa,Ya,Nab,Xb,Yb,Xc,Yc,Ncd,Xd,Yd) :-
        tell(user),                          /* 'before' compared with 'after' */
        write('0v/D (Type 4) bdry being plotted between edges '),
        write(Nab),write(' and '),write(Ncd),nl,
        abolish(bdry,1),
        optimal_path([Xb,Yb,c(C1),Xp1,Yp1|OPp1]),
        counterclockwise([Xb,Yb,c(C1),Xp1,Yp1|OPp1]),
        path_length([Xp1,Yp1|OPp1],D1),
        optimal_path([Xc,Yc,c(C2),Xp2,Yp2|OPp2]),
        clockwise([Xc,Yc,c(C2),Xp2,Yp2|OPp2]),
        path_length([Xp2,Yp2|OPp2],D2),
        not(same(D1,D2)),
        distance(Xp1,Yp1,Xp2,Yp2,D3),
        distance(Xc,Yc,Xp1,Yp1,Dcp1),
        distance(Xc,Yc,Xp2,Yp2,Dcp2),
        distance(Xb,Yb,Xp1,Yp1,Dbp1),
        distance(Xb,Yb,Xp2,Yp2,Dbp2),
        pi(Pi),
        Cos1 is (D3^2+Dcp2^2-Dcp1^2)/(2*D3*Dcp2),
        arccos(Cos1,Piover2plusAlpha),
        A is Piover2plusAlpha - (Pi/2),
        Cos2 is (D3^2+Dbp1^2-Dbp2^2)/(2*D3*Dbp1),
        arccos(Cos2,Piover2plusBeta),
        B is Piover2plusBeta - (Pi/2),
        Tc is asin(Ce/Ci),
        compute_angle_of_rotation(Xa,Ya,Xb,Yb,RotAngle),
        calc_bdry_pt_0visD(A,B,Tc,Xp1,Yp1,D1,D2,D3,0,X2bdry,Y2bdry),
        calc_bdry_pt_0visD(A,B,Tc,Xp1,Yp1,D1,D2,D3,Dbp1,X1bdry,Y1bdry),
        abolish(bdry,1),
        assert(bdry([X1bdry,Y1bdry,X2bdry,Y2bdry,Xp1,Yp1])),
        invert_bdry,
        rotate2_bdry(RotAngle),
        bdry([X1,Y1,X2,Y2]),
        correct_error_in_opp_edge(Xb,Yb,Xc,Yc,X1,Y1,X1r,Y1r),
```

```
        assert(initbdry([Nab,Ncd],[X1r,Y1r,X2,Y2])), !.

plot_Ovis_bdry(_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_) :-
        tell(user), write('            Bdry does not exist.'),nl, !.

calc_bdry_pt_OvisD(A,B,Tc,Xo,Yo,D1,D2,D3,X1,X,Y) :-
        TcplusA is Tc + A,
        F1 is sin(A) + cos(TcplusA)*sin(Tc),
        T1 is cos(Tc)*(sin(B)-sin(Tc)*cos(TcplusA)) + cos(Tc+A+B)*F1,
        T2 is cos(Tc) - sin(TcplusA)*F1,
        T3 is cos(TcplusA)*cos(Tc)*sin(Tc),
        Denom is cos(TcplusA)*cos(Tc)+cos(Tc+B)*cos(Tc)-sin(2*Tc+A+B)*F1,
        X2 is (X1*T1 + D3*T2 + (D2-D1)*T3) / Denom,
        X is Xo - (X1+X2*sin(Tc)),
        Y is Yo - X2*cos(Tc), !.

calc_bdry_pt_OvisM(A,B,Tc,Xo,Yo,Z,D1,D2,D3,X1,X,Y) :-
        T1 is sin(A+B)+sin(Tc)*cos(Tc+A+B)-cos(Tc)*sin(Tc),
        T2 is D3*(sin(A)+sin(Tc)*cos(Tc+A)),
        T3 is D2*cos(Tc)*sin(Tc),
        T4 is Z*sin(Tc)*cos(Tc),
        T5 is cos(Tc)-cos(Tc-A-B)+sin(Tc)*sin(A+B),
        X2 is (X1*T1 + T2 - T3 - T4)/T5,
        X is Xo - (X1+X2*sin(Tc)),
        Y is Yo + X2*cos(Tc), !.

/* If edges are connected then pt B = pt C  and bdry */
/* should start exactly at pt P1. */
correct_error_in_conn_edges(Xb,Yb,Xb,Yb,Xp1,Yp1,_,_,Xp1,Yp1).
/* If edges are not connected use the */
/* bdry point as calculated. */
correct_error_in_conn_edges(_,_,_,_,_,_,X1,Y1,X1,Y1).

/* If edges are part of the opposite edge, then */
/* bdry should start exactly at the opposite point. */
correct_error_in_opp_edge(Xb,Yb,Xc,Yc,_,_,Xopp,Yopp) :-
        opposite_point_minus(Xb,Yb),
        opposite_point_plus(Xc,Yc),
        opposite_point(Xopp,Yopp).
/* If not, then leave start of bdry unchanged.  */
correct_error_in_opp_edge(_,_,_,_,X,Y,X,Y).



/*****************************************************************************
 *
 *   plotoebdry computes HCA Interior Opposite-Edge Boundaries (Type 5)
 *   Assumes no shortcutting!
 *
 *   Updated 12 Jan 89.
 *
 */

plotoebdry(Ci,Ce,Xa,Ya,Xb,Yb) :-
        plotoebdry(Ci,Ce,Xa,Ya,Xb,Yb,B).
plotoebdry(Ci,Ce,Xa,Ya,Xb,Yb,Bdry) :-
        optimal_path([Xb,Yb|_],D1),
        optimal_path([Xa,Ya|_],D2),
        distance(Xa,Ya,Xb,Yb,D3),
        compute_angle_of_rotation(Xb,Yb,Xa,Ya,RotAngle),
        plot2oebdry(Ci,Ce,D1,D2,D3,Xb,Yb,RotAngle,Bdry),
        /* tell(user),write([Xa,Ya,Xb,Yb]),nl,*/
        !.
```

```
plot2oebdry(IntCost,ExtCost,D1,D2,D3,Vx,Vy,Angle,RevBdry) :-
        initialize_for5,
        tell(user), nl,nl,
        /* write('HCA Int Opp-Edge (Type 5) bdry being plotted for edge '), */
        Tc is asin(ExtCost/IntCost),
        calc_oebdry_pts(Tc,D1,D2,D3,Vx,Vy,Angle),
        rotate_bdry(Angle),
        retract(bdry(Bdry)),
        remove_last_pt(Bdry,RevBdry),
        truncate_off_map(RevBdry,FinalBdry),
        assert(initbdry(oe,FinalBdry)),
        /* tell(bdry_out),write('           Type 5 '),nl,
        write_to_bdry_file(bdry,RevBdry),
        output_to_figure_file, */
        !.

calc_oebdry_pts(Tc,D1,D2,D3,Vx,Vy,Angle) :-
        X11 is (D3+D2-D1)/2,
        X22 is X11/sin(Tc),
        Xa is Vx + X11,
        Ya is Vy,
        Xb is Vx + X22*sin(Tc),
        Yb is Vy + X22*cos(Tc),
        assert(bdry([Xa,Ya,Xb,Yb,Vx,Vy])), !.

initialize_for5 :-
        abolish(bdry,1), !.

/* First check whether bdry starts on the map.  If so, call trunc..2 */
/* if not, call trunc..3 */
truncate_off_map([X1,Y1,X,Y|B],B2) :-
        minX(MinX),minY(MinY),
        maxX(MaxX),maxY(MaxY),
        DelX is MaxX - MinX,
        X1 > MinX, X1 < MaxX, Y1 > MinY, Y1 < MaxY,      /* starts ON the map.*/
        truncate_off_map2([X1,Y1,X,Y|B],B2), !.
truncate_off_map([X1,Y1,X,Y|B],B2) :-                    /* starts OFF the map.*/
        truncate_off_map3([X1,Y1,X,Y|B],B2), !.
truncate_off_map2([X1,Y1,X,Y|B],[X1,Y1|B2]) :-           /* Assumes that Bdry */
        minX(MinX),minY(MinY),                           /* starts ON the map.*/
        maxX(MaxX),maxY(MaxY),
        DelX is MaxX - MinX,
        MinX2 is MinX - 0.1*DelX,
        MinY2 is MinY - 0.1*DelX,
        MaxX2 is MaxX + 0.1*DelX,
        MaxY2 is MaxY + 0.1*DelX,
        X > MinX2, X < MaxX2, Y > MinY2, Y < MaxY2,
        truncate_off_map2([X,Y|B],B2), !.
truncate_off_map2([X1,Y1,X,Y|B],[X1,Y1,X,Y]) :-  !.
truncate_off_map2([X,Y],[X,Y]) :-  !.
truncate_off_map3([X1,Y1,X,Y],[X1,Y1,X,Y]).   /* Bdry is entirely off the map */
                                          /* except perhaps for the last pt. */
truncate_off_map3([X1,Y1,X,Y|B],B2) :-          /* Bdry starts off the map.*/
        truncate_off_map([X,Y|B],B2), !.

assert_shortcut_flag(Xv,Yv,Xv,Yv).
assert_shortcut_flag(Xv,Yv,Xp1,Yp1) :-
        assert(shortcut(Xv,Yv,Xp1,Yp1)), !.

clockwise([X1,Y1,c(C),Xi,Yi|OP]) :-
        region_elist(R),
        get_XYcw_edge(X1,Y1,R,[Xa,Ya,Xb,Yb]),
        on_ray(Xi,Yi,X1,Y1,Xb,Yb), !.
counterclockwise([X1,Y1,c(C),Xi,Yi|OP]) :-
```

```
        region_elist(R),
        get_XYccw_edge(X1,Y1,R,[Xa,Ya,Xb,Yb]),
        on_ray(Xi,Yi,X1,Y1,Xa,Ya), !.

get_XYcw_edge(X1,Y1,[Xa,Ya,_,_,Xb,Yb|R],[Xa,Ya,Xb,Yb]) :-
        on_ray(X1,Y1,Xb,Yb,Xa,Ya), !.
get_XYcw_edge(X1,Y1,[Xa,Ya,_,_,Xb,Yb|R],Edge) :-
        get_XYcw_edge(X1,Y1,R,Edge), !.

get_XYccw_edge(X1,Y1,[Xa,Ya,_,_,Xb,Yb|R],[Xa,Ya,Xb,Yb]) :-
        on_ray(X1,Y1,Xa,Ya,Xb,Yb), !.
get_XYccw_edge(X1,Y1,[Xa,Ya,_,_,Xb,Yb|R],Edge) :-
        get_XYccw_edge(X1,Y1,R,Edge), !.
```

```
/*************************************************************************
 *************************************************************************
 *
 * File "boundary_join" or "bj"
 *
 * Updated 30 Jan 89
 *
 * "bdry_join" truncates boundaries and joins them together into
 * a network of the active boundaries inside a homogeneous-cost region.
 *
 *************************************************************************
 *************************************************************************/



/*************************************************************************/
/*********************  Top-level predicate  *************************/
/*************************************************************************/

bdry_join :-
        initialize,
        first_level_bdrys(A1),
        assert(old_bdry_set([])),
        assert(current_bdry_set(A1)),
        retract(current_bdry_set(Acur)),     /* start of while-not-done loop */
        retract_cut(old_bdry_set(Aold)),
        not(same_set(Acur,Aold)),
        assert(old_bdry_set(Acur)),
        next_level_bdrys(Acur,Anew),
        assert(current_bdry_set(Anew)),
        done(Anew),                          /* end of while-not-done loop */
        get_final_bdrys(Afinal),
        output(Afinal),
        cleanup, halt, !.
bdry_join :-
        tell(user), nl,nl,
        write('   Checking for center shortcutting'),nl,nl,
        elim_incomplete_trees,
        get_final_bdrys(Afinal),
        bdry_edge_intersections(Afinal,BEI),
        find_exact_opposite_pt(BEI,P),
        recurse_unless_done(P),
        output(Afinal),
        cleanup, halt, !.
bdry_join :-
        tell(user), nl,nl,
        write('   ERROR in ''bdry-join'': doesn''t converge'),nl,nl, !.


/*************************************************************************/
/*********************  Second-level predicates  *********************/
/*************************************************************************/

initialize :-
        tell(user),
        nl,nl, write('Boundaries being joined:'), nl,nl,
        abolish(ctr,1),
        assert(ctr(1)).

first_level_bdrys(A2) :-
        number_of_edges(N),
        index_list_ItoJ(1,N,IndexList0),
        cons(IndexList0,[[1,2]],IndexList),
```

```
        assert(indices(IndexList)),
        first_level_bdrys1(A1,N),
        order_indices(A1,A2), !.

next_level_bdrys(A1,A3) :-
        reset(A1,A2),
        propogate_next_level_bdrys(A2),
        get_active_bdry_set(A3), !.

done(A) :-
        one_bdry_tree(A),
        tell(user),nl,nl,nl,
        write_to_screen('           DONE - single bdry-tree'),nl,nl,nl, !.

get_final_bdrys(A) :-
        get_tbdrys(A), !.

bdry_edge_intersections([],[]).
bdry_edge_intersections([[[I,J],B,LPt]|L],[[[I,J],B,LPt]|BEI]) :-
        edge_bdry_intersection(K,[I,J],LPt),
        bdry_edge_intersections(L,BEI), !.
bdry_edge_intersections([[[I,J],B,LPt]|L],BEI) :-
        bdry_edge_intersections(L,BEI), !.


elim_incomplete_trees :-
        tree([I,J],L,R),
        not(complete_tree(tree([I,J],L,R))),
        eliminate_tree_tbdrys(tree([I,J],L,R)),
        fail, !.
elim_incomplete_trees :- !.

find_exact_opposite_pt(BEI,P) :-
        find_exact_opposite_pt1(BEI),
        opposite_edge(OE),
        new_opp_pt(OE,OE,P),!.
find_exact_opposite_pt1([]).
find_exact_opposite_pt1([[[I,J],B,[LX,LY]]|BEI]) :-
        optimal_path([LX,LY,c(C),X2,Y2|OP]),
        update_opp_edge(I,J,[LX,LY,c(C),X2,Y2|OP]),
        find_exact_opposite_pt1(BEI), !.

recurse_unless_done([Xopp2,Yopp2]) :-
        not(first_pass_done),
        opposite_point(Xopp1,Yopp1),
        not(same([Xopp1,Yopp1],[Xopp2,Yopp2])),
        retract(opposite_point(Xopp1,Yopp1)),
        assert(opposite_point(Xopp2,Yopp2)),
        assert(first_pass_done),
        cleanup2,
        bg2, !.
recurse_unless_done([Xopp2,Yopp2]) :-
        tell(user),nl,nl,nl,
        write_to_screen('           DONE - Finished Second Pass'),nl,nl,nl, !.

output(A) :-
        write_heading,
        write_bdrys_to_file(hca_opm,A), nl, !.

cleanup :-
        abolish(ctr,1),
        abolish(tbdry,5),
        abolish(current_bdry_set,1),
        abolish(bdry_list,1),
```

```
        abolish(bdry_intersection,5),
        abolish(initbdry,2),
        abolish(tree,3),
        tell(user),
        nl,nl,
        write('Boundary generation complete:  results in file ''hca_opm'''),
        nl,nl, !.
cleanup2 :-
        abolish(ctr,1),
        abolish(tbdry,5),
        abolish(current_bdry_set,1),
        abolish(bdry_list,1),
        abolish(initbdry,2),
        abolish(region_elist,1),
        abolish(pseudo_optimal_path,1),
        abolish(tree,3),
        tell(user),
        nl,nl,write('Pass Two beginning'),nl,nl, !.


/*************************************************************************/
/************  "initialization" subordinate predicates  *************/
/*************************************************************************/

/* Assert the points at which bdrys are 'anchored' to the region edges */
assert_anchors :-
        number_of_edges(N),
        index_list_ItoJ(1,N,IndexList1),
        assert_anchors(IndexList), !.
assert_anchors([]) :- !.
assert_anchors([[I,J]|L]) :-
        initbdry([I,J],[X,Y|B]),
        assert(anchor(X,Y)),
        assert_anchors(L), !.

reset(A,A1) :-
        abolish(tbdry,5),
        abolish(edge_int_pt,3),
        order_indices(A,A1),
        reassert_tbdrys(old,A1,1),
        abolish(ctr,1),
        assert(ctr(1)),!.



/*************************************************************************/
/**********  "first-level-bdrys" subordinate predicates  ************/
/*************************************************************************/

first_level_bdrys1(A,N) :-
        retract(indices(IndexList)),
        truncate_lst_level_bdrys(IndexList,N),
        retract_all_and_rtn_shortest_tbdrys(ShortBdrys),
        matching_pairs(ShortBdrys,PairedBdrys),
        bdry_edge_intersections(ShortBdrys,EdgeIntBdrys),
        set_subtraction(EdgeIntBdrys,PairedBdrys,EdgeIntBdrys2),
        cons(EdgeIntBdrys2,PairedBdrys,ActiveBdrys),
        reassert_tbdrys(old,ActiveBdrys,1),
        Nminus1 is N - 1,
        not(list_length(ActiveBdrys,Nminus1)),
        not(list_length(ActiveBdrys,N)),
        index_list(ActiveBdrys,IndexL1),
        set_subtraction(IndexList,IndexL1,IndexL2),
        /* complement_index_list(N,IndexL1,IndexL2), */
```

```
        not(same_set(IndexList,IndexL2)),       /* If same, no new bdry pairs */
        assert(indices(IndexL2)),
        first_level_bdrys1(A,N), !.
first_level_bdrys1(A,N) :-
        assert_singles(1,N),
        get_tbdrys(A), !.

matching_pairs([FirstB|Rest],RevRest) :-
        matching_pairs1([FirstB|Rest],FirstB,RevRest), !.

matching_pairs1([],_,[]) :-  !.
matching_pairs1([[IDlast,Blast,LPtfirst]],[IDfirst,Bfirst,LPtfirst],
        [[IDlast,Blast,LPtfirst],[IDfirst,Bfirst,LPtfirst]]) :- !.
matching_pairs1([[IDlast,Blast,LPtlast]],[IDfirst,Bfirst,LPtfirst],[]) :- !.
matching_pairs1([[ID1,B1,LPt1],[ID2,B2,LPt1]|Rest],Bfirst,
        [[ID1,B1,LPt1],[ID2,B2,LPt1]|RevRest]) :-
        matching_pairs1(Rest,Bfirst,RevRest), !.
matching_pairs1([B1,B2|Rest],Bfirst,RevRest) :-
        matching_pairs1([B2|Rest],Bfirst,RevRest), !.

truncate_1st_level_bdrys([[_,_]],N) :- !.             /* Base case */
truncate_1st_level_bdrys([[N,1],[1,2]],N) :-        /* Last pair of bdrys: */
        initbdry([1,N],[X1,Y1|B1]),           /* succeeds if they intersect. */
        initbdry([1,2],[X2,Y2|B2]),
        bdry_intersection([X1,Y1|B1],[X2,Y2|B2],IntPt,B1trunc,B2trunc),
        get_counter_and_increment(C0),
        get_counter_and_increment(C1),
        assert(tbdry(new,C0,[N,1],B1trunc,IntPt)),
        assert(tbdry(new,C1,[1,2],B2trunc,IntPt)),
        region_elist(R),
        truncate_bdry_and_edges([1,N],[X1,Y1|B1],R),
        truncate_1st_level_bdrys([[_,_]],N), !.
truncate_1st_level_bdrys([[Nminus1,N],[N,1]|Rest],N) :-
        Nminus1 is N-1,                        /* Next to Last pair of bdrys: */
        initbdry([Nminus1,N],[X1,Y1|B1]),      /* succeeds if they intersect. */
        initbdry([1,N],[X2,Y2|B2]),
        bdry_intersection([X1,Y1|B1],[X2,Y2|B2],IntPt,B1trunc,B2trunc),
        get_counter_and_increment(C0),
        get_counter_and_increment(C1),
        assert(tbdry(new,C0,[Nminus1,N],B1trunc,IntPt)),
        assert(tbdry(new,C1,[N,1],B2trunc,IntPt)),
        region_elist(R),
        truncate_bdry_and_edges([N,Nminus1],[X1,Y1|B1],R),
        truncate_1st_level_bdrys([[N,1]|Rest],N), !.
truncate_1st_level_bdrys([[I,J],[J,K]|Rest],N) :- /* Succeeds if bdrys are*/
        initbdry([I,J],[X1,Y1|B1]),                /* adjacent and intersect*/
        initbdry([J,K],[X2,Y2|B2]),
        bdry_intersection([X1,Y1|B1],[X2,Y2|B2],IntPt,B1trunc,B2trunc),
        get_counter_and_increment(C0),
        get_counter_and_increment(C1),
        assert(tbdry(new,C0,[I,J],B1trunc,IntPt)),
        assert(tbdry(new,C1,[J,K],B2trunc,IntPt)),
        region_elist(R),
        truncate_bdry_and_edges([I,J],[X1,Y1|B1],R),
        truncate_1st_level_bdrys([[J,K]|Rest],N), !.
truncate_1st_level_bdrys([[I,J],[K,L]|Rest],N) :-
        ordered(I,J,I1,J1),                          /* Recurses if previous  */
        initbdry([I1,J1],[X1,Y1|B1]),                /* adjacent and intersect*/
        region_elist(R),                             /* rules have failed.    */
        truncate_bdry_and_edges([I1,J1],[X1,Y1|B1],R),
        truncate_1st_level_bdrys([[K,L]|Rest],N), !.

/* Asserts a temp tbdry which stops at the region edge if initbdry(I,J) */
/* intersects a region opposite edge.  Always succeeds.  Also asserts   */
```

```
/* 'edge_bdry_intersection(K,[I,J],[X,Y])' for each intersection point. */
truncate_bdry_and_edges([I,J],[X,Y|B],[X1,Y1,h(Q),K,X2,Y2|R]) :-
        not(bdry_starts_at_edge(I,J,K)),
        bdry_intersection([X,Y|B],[X1,Y1,X2,Y2],IntPt,Btrunc,Bltrunc),
        get_counter_and_increment(C0),
        assert(edge_bdry_intersection(K,[I,J],IntPt)),
        assert(tbdry(temp,C0,[I,J],Btrunc,IntPt)), !.
truncate_bdry_and_edges([I,J],[X,Y|B],[X1,Y1,_,K,X2,Y2|R]) :-
        truncate_bdry_and_edges([I,J],[X,Y|B],R), !.
truncate_bdry_and_edges([I,J],[X,Y|B],[]) :- !.

assert_singles(I,N) :-                  /* If any vertex's 1st-level bdry */
        Iplus1 is I + 1,                /* has not yet been generated, do */
        Iplus1 < N,                     /* so now. */
        tbdry(_,_,[I,Iplus1],_,_),
        assert_singles(Iplus1,N), !.
assert_singles(I,N) :-
        Iplus1 is I + 1,
        Iplus1 < N,
        not(tbdry(_,_,[I,Iplus1],_,_)),
        initbdry([I,Iplus1],B),
        get_counter_and_increment(Ctr),
        assert(tbdry(new,Ctr,[I,Iplus1],B,[])),
        assert_singles(Iplus1,N), !.
assert_singles(I,N) :-
        Iplus1 is I + 1,
        Iplus1 = N,
        tbdry(_,_,[N,1],_,_), !.
assert_singles(I,N) :-
        Iplus1 is I + 1,
        Iplus1 = N,
        not(tbdry(_,_,[N,1],_,_)),
        initbdry([1,N],B),
        assert(tbdry(new,Ctr,[N,1],B,[])), !.



/*************************************************************************/
/*************  "next-level-bdrys" subordinate predicates  **********/
/*************************************************************************/

propogate_next_level_bdrys([]) :- !.
propogate_next_level_bdrys([[[I,J],B1,[LX,LY]]|A]) :-
        tbdry(_,_,[K,L],B2,[LX,LY]),            /* Previously connected at end */
        adjacent_bdrys(I,J,K,L,I1,J1,K1,L1),
        not(same(I1,L1)),                       /* Not same bdry */
        ordered(I1,L1,I2,L2),
        not(tbdry(_,_,[I2,L2],[LX,LY|_],_)),    /* Not previously asserted */
        ordered(I1,J1,I3,J3),                   /* Use indices in order */
        initbdry([I3,J3],B1Full),
        ordered(I1,L1,I4,L4),
        initbdry([I4,L4],B12),
        bdry_intersection(B1Full,B12,[IntX,IntY],_,B12trunc),
        within_tolerance(LX,LY,IntX,IntY),
        get_correct_half_of_bdry(B1,B2,B12,[LX,LY],B12trunc,[X12,Y12|B12cor]),
        get_last_pt([X12,Y12|B12cor],B12Xlast,B12Ylast),
        get_counter_and_increment(C1),
        assert(tbdry(new,C1,[I2,L2],[LX,LY|B12cor],[B12Xlast,B12Ylast])),
        propogate_next_level_bdrys(A), !.
propogate_next_level_bdrys([[[I,J],B,LPt]|A]) :-   /* Disregard bdry which is */
        propogate_next_level_bdrys(A), !.       /* paired with another bdry or */
                                                /* intersects a region edge.   */
propogate_next_level_bdrys([[[I,J],B1,[]]|A]) :-    /* Disregard single bdry */
        propogate_next_level_bdrys(A), !.
```

```
get_correct_half_of_bdry([X1,Y1|B1],[X2,Y2|B2],[X12,Y12|B12],
        [Xi,Yi],B12tr,[Xi,Yi|B12corr]) :-        /* Intersect a line from B1 */
        X1test is Xi+((X1-Xi)/20),      /* to B2 drawn just inside their pt of*/
        Y1test is Yi+((Y1-Yi)/20),      /* intersection, with the new bdry. */
        X2test is Xi+((X2-Xi)/20),      /* If no inters, bdry is outside B1 */
        Y2test is Yi+((Y2-Yi)/20),      /* and B2, so this is correct half. */
        not(bdry_intersection_exact([X1test,Y1test,X2test,Y2test],B12tr,_,_,_)),
        reverse_path_list(B12tr,[_,_|B12corr]), !.     /* but reversed.   */
get_correct_half_of_bdry(B1,_,B12,
        [Xi,Yi],_,[Xi,Yi|B12otherhalf]) :-       /* Otherwise get the other */
        reverse_path_list(B12,B12Rev),           /* half of new bdry.       */
        bdry_intersection(B1,B12Rev,_,_,B12trunc),
        reverse_path_list(B12trunc,[_,_|B12otherhalf]), !.

get_active_bdry_set(_) :-
        tbdry(new,_,[I,J],B1,LPt1),
        intersect_with_candidate_bdrys(I,J,B1,LPt1),
        fail, !.
get_active_bdry_set(A) :-
        retract_all_and_rtn_shortest_tbdrys(ShortBdrys),
        while_changing_reassert_tbdrys(ShortBdrys,1,_,_),
        reset_last_pts,
        get_tbdrys(A), !.

intersect_with_candidate_bdrys(I,J,[X1,Y1|B1],LPt1) :-
        get_tbdryIJorJI(F,_,[I,K],B2,LPt2),
        not(same(F,temp)),                              /* not a temporary bdry */
        not(same([X1,Y1],LPt2)),                        /* not a child of B1    */
        ordered(I,J,I1,J1),
        ordered(I,K,I2,K2),
        not(same([I1,J1],[I2,K2])),                     /* not the same as B1   */
        interior_intersection([X1,Y1|B1],LPt1,B2,LPt2,IntPt,B1trunc,B2trunc),
        not(asserted_tbdry([I2,K2],IntPt)), /*If B1 intersects*/
        get_counter_and_increment(C0),                  /* the candidate, then  */
        assert(tbdry(temp,C0,[I2,K2],B2trunc,IntPt)), /* assert both as temps */
        not(asserted_tbdry([I1,J1],IntPt)), /*if not asserted */
        get_counter_and_increment(C1),
        assert(tbdry(temp,C1,[I1,J1],B1trunc,IntPt)),
        fail, !.
intersect_with_candidate_bdrys(I,J,[X1,Y1|B1],LPt1) :-
        get_tbdryIJorJI(F,_,[J,L],B2,LPt2),
        not(same(F,temp)),                              /* not a temporary bdry */
        not(same([X1,Y1],LPt2)),                        /* not a child of B1    */
        ordered(I,J,I1,J1),
        ordered(J,L,J2,L2),
        not(same([I1,J1],[J2,L2])),                     /* not the same as B1   */
        interior_intersection([X1,Y1|B1],LPt1,B2,LPt2,IntPt,B1trunc,B2trunc),
        not(asserted_tbdry([J2,L2],IntPt)), /*If B1 intersects*/
        get_counter_and_increment(C0),                  /* the candidate, then  */
        assert(tbdry(temp,C0,[J2,L2],B2trunc,IntPt)), /* assert both as temps */
        not(asserted_tbdry([I1,J1],IntPt)),  /* if not asserted*/
        get_counter_and_increment(C1),
        assert(tbdry(temp,C1,[I1,J1],B1trunc,IntPt)),
        fail, !.
intersect_with_candidate_bdrys(I,J,[X1,Y1|B1],LPt1) :- /* Intersect bdry with */
        region_elist(R),                                /* region edges.       */
        truncate_bdry_and_edges([I,J],[X1,Y1|B1],R), !.

asserted_tbdry([I,J],LPt) :-            /* tbdry is already asserted */
        tbdry(_,_,[I,J],_,LPt), !.
asserted_tbdry([I,J],[Xi1,Yi1]) :-      /* tbdry with appx= last pt */
        tbdry(_,_,[I,J],_,[Xi2,Yi2]),          /* is already asserted */
        within_tolerance(Xi1,Yi1,Xi2,Yi2), !.
```

```
retract_all_and_rtn_shortest_tbdrys([[[I,J],BminD,LPtminD]|Rest]) :-
        tbdry(_,_,[I,J],[X,Y|B],_),                /* Retract all IJ bdrys  */
        retract_IJ_bdrys(I,J,Bdrys),
        get_shortest_tbdry(Bdrys,_,BminD,LPtminD),
        retract_all_and_rtn_shortest_tbdrys(Rest), !.
retract_all_and_rtn_shortest_tbdrys([]) :- !.

retract_IJ_bdrys(I,J,[[[X,Y|B],LPt]|Bdrys]) :-  /* Retract all bdrys with */
        retract(tbdry(_,_,[I,J],[X,Y|B],LPt)),   /* index I,J, and return  */
        retract_IJ_bdrys(I,J,Bdrys).             /* them in a list */
retract_IJ_bdrys(_,_,[]) :- !.
xretract_IJ_bdrys(I,J,X,Y,[[[X,Y|B],LPt]|Bdrys]) :-  /* Retract all bdrys */
        retract(tbdry(_,_,[I,J],[X,Y|B],LPt)),       /* with index I,J which */
        retract_IJ_bdrys(I,J,X,Y,Bdrys).             /* have same starting pt */
xretract_IJ_bdrys(_,_,_,_,[]) :- !.                  /* & rtn them in a list */

get_shortest_tbdry([],100000,_,_) :- !.
get_shortest_tbdry([[B,LPt]|Bdrys],NewMinD,NewB,NewLPt) :-
        path_length(B,D),
        get_shortest_tbdry(Bdrys,MinD,BminD,LPtminD),
        get_minD_and_B(D,B,LPt,MinD,BminD,LPtminD,NewMinD,NewB,NewLPt), !.
get_minD_and_B(D,B,LPtB,MinD,BminD,LPtminD,D,B,LPtB) :- D < MinD, !.
get_minD_and_B(D,B,LPtB,MinD,BminD,LPtminD,MinD,BminD,LPtminD) :- !.

while_changing_reassert_tbdrys(Set1,Ctr1,Set2,Ctr2) :-
        reassert_connected_tbdrys(Set1,Ctr1,Set2,Ctr2),
        not(same_set(Set1,Set2)),
        while_changing_reassert_tbdrys(Set2,Ctr2,_,_), !.
while_changing_reassert_tbdrys(Set1,Ctr1,Set2,Ctr2) :-
        same_set(Set1,Set2), !.
while_changing_reassert_tbdrys(Set1,Ctr1,Set2,Ctr2) :-
        write_to_screen('Error in -reassert_tbdrys- '),nl,!.

reassert_connected_tbdrys([],Ctr,[],Ctr) :- !.
reassert_connected_tbdrys([[[I,J],B,LPt]|ActSet],Ctr,InteriorBs,Ctr2) :-
        connected_to_an_anchor([I,J],B),
        ordered(I,J,K,L),
        assert(tbdry(old,Ctr,[I,J],B,LPt)),
        Cplus1 is Ctr+1,
        reassert_connected_tbdrys(ActSet,Cplus1,InteriorBs,Ctr2).
reassert_connected_tbdrys([[[I,J],B,LPt]|ActSet],Ctr,
        [[[I,J],B,LPt]|InteriorBs],Ctr2) :-
        reassert_connected_tbdrys(ActSet,Ctr,InteriorBs,Ctr2).

connected_to_an_anchor([I,J],B0) :-          /* Bdry starts at a region edge */
        adjacent_edges(I,J), !.
connected_to_an_anchor([I,J],[X1,Y1|B]) :-   /* Bdry starts at last pt of */
        tbdry(_,_,[K,L],B2,[X1,Y1]),         /* another anchored bdry.    */
        connected_to_an_anchor([K,L],B2), !.
connected_to_an_anchor([I,J],[X1,Y1|B]) :-   /* Bdry starts at first pt of */
        tbdry(_,_,[K,L],[X1,Y1|B2],_),       /* another anchored bdry.    */
        connected_to_an_anchor([K,L],B2), !.



/*************************************************************************/
/*****************  "done" subordinate predicates  ***********************/
/*************************************************************************/

/* Example 'tree' fact (indented for clarity only - root is at left):

   tree([7,7], tree([7,3], tree([7,2],  tree([7,1],'[]','[]'),
```

```
                                        tree([1,2],'[]','[]')),
                                tree([2,3],'[]','[]')),
                        tree([3,7], tree([3,4],'[]','[]'),
                                    tree([4,7],  tree([4,5],'[]','[]'),
                                                 tree([5,7],tree([5,6],'[]','[]'),
                                                            tree([6,7],'[]','[]'))))).

                                [1,7]
                               /     \
                              /       \
                          [7,3]         [3,7]
                         /     \       /     \
                    [7,2]       [2,3]  [3,4]   [4,7]
                   /     \                     /     \
              [7,1]       [1,2]             [4,5]     [5,7]
                                                     /     \
                                                 [5,6]     [6,7]          */

one_bdry_tree([[[I,J],B,LPt]|A]) :-
        abolish(tree,3),
        number_of_edges(N),
        assert_leaf_trees(1,N),
        while_changing_combine_trees(N),
        number_of_trees(NT),
        NT == 1, !.

while_changing_combine_trees(N) :-
        abolish(number_of_trees,1),
        assert(number_of_trees(N)),
        retract(number_of_trees(PrevNT)),
        combine_trees(1,N,NT),
        assert(number_of_trees(NT)),
        PrevNT == NT, !.              /* fails here until no longer changing */

combine_trees(I,N,NumTrees) :-
        I < N,
        tree([I,J],Left1,Right1),
        tree([J,K],Left2,Right2),
        not(same(I,K)),
        succeed_if_joined(I,J,K),
        retract(tree([I,J],Left1,Right1)),
        retract(tree([J,K],Left2,Right2)),
        assert(tree([I,K],tree([I,J],Left1,Right1),tree([J,K],Left2,Right2))),
        Iplus1 is I + 1,
        combine_trees(Iplus1,N,NumTrees), !.
combine_trees(I,N,NumTrees) :-              /* Final case, where tree([I,J],,) */
        tree([I,J],Left1,Right1),           /* is combined with tree([J,I],,)  */
        tree([J,I],Left2,Right2),
        not(same(Left1,Left2)),
        succeed_if_joined(I,J,I),  /* TEMP: always succeeds. Won't always */
        retract(tree([I,J],Left1,Right1)),        /* succeed for center-s/c */
        retract(tree([J,I],Left2,Right2)),
        assert(tree([I,I],tree([I,J],Left1,Right1),tree([J,I],Left2,Right2))),
        count_trees(NumTrees), !.
combine_trees(I,N,1) :-                 /* Last iter. of while-changing loop, */
        tree([I,I],Left1,Right1), !.    /* where single tree is tree([I,I],..) */
combine_trees(I,N,NumTrees) :-
        I < N,
        Iplus1 is I + 1,
        combine_trees(Iplus1,N,NumTrees), !.
combine_trees(N,N,NumTrees) :-
        tree([N,J],Left1,Right1),
        tree([J,K],Left2,Right2),
        not(same(N,K)),
```

```
        succeed_if_joined(N,J,K),
        retract(tree([N,J],Left1,Right1)),
        retract(tree([J,K],Left2,Right2)),
        assert(tree([N,K],tree([N,J],Left1,Right1),tree([J,K],Left2,Right2))),
        count_trees(NumTrees), !.
combine_trees(N,N,NumTrees) :-        /* Base case for all but last iteration */
        count_trees(NumTrees), !.

count_trees(NumTrees) :-
        abolish(count,1),
        assert(count(0)),
        tree(_,_,_),
        retract_cut(count(C)),
        NumTrees is C + 1,
        assert(count(NumTrees)),
        fail, !.
count_trees(NumTrees) :-
        retract(count(NumTrees)), !.

assert_leaf_trees(I,N) :-
        I < N,
        Iplus1 is I + 1,
        assert(tree([I,Iplus1],[],[])),
        assert_leaf_trees(Iplus1,N), !.
assert_leaf_trees(N,N) :-
        assert(tree([N,1],[],[])), !.

/* Succeeds if bdrys I,J , J,K , and I,K are joined at one point */
succeed_if_joined(I,J,I).
succeed_if_joined(I,I,J).
succeed_if_joined(J,I,I).
succeed_if_joined(I,J,K) :-
        ordered(I,J,I1,J1),
        tbdry(_,_,[I1,J1],[Xij,Yij|Bij],[LXij,LYij]),
        ordered(J,K,J2,K2),
        tbdry(_,_,[J2,K2],[Xjk,Yjk|Bjk],[LXjk,LYjk]),
        ordered(I,K,I3,K3),
        tbdry(_,_,[I3,K3],[Xik,Yik|Bik],[LXik,LYik]),
        match_3_pts(Xij,Yij,LXij,LYij,Xjk,Yjk,LXjk,LYjk,Xik,Yik,LXik,LYik),!.

/* Succeeds if there is a match among any permutation of the three */
/* pairs of points; First check for exact matches: */
match_3_pts(X,Y,_,_,X,Y,_,_,X,Y,_,_).
match_3_pts(_,_,X,Y,X,Y,_,_,X,Y,_,_).
match_3_pts(X,Y,_,_,_,_,X,Y,X,Y,_,_).
match_3_pts(_,_,X,Y,_,_,X,Y,X,Y,_,_).
match_3_pts(X,Y,_,_,X,Y,_,_,_,_,X,Y).
match_3_pts(_,_,X,Y,X,Y,_,_,_,_,X,Y).
match_3_pts(X,Y,_,_,_,_,X,Y,_,_,X,Y).
match_3_pts(_,_,X,Y,_,_,X,Y,_,_,X,Y).
/* If no exact match, check for approximate matches: */
match_3_pts(X1,Y1,_,_,X2,Y2,_,_,X3,Y3,_,_) :-
        within_tolerance(X1,Y1,X2,Y2),
        within_tolerance(X1,Y1,X3,Y3),
        within_tolerance(X2,Y2,X3,Y3), !.
match_3_pts(_,_,X1,Y1,X2,Y2,_,_,X3,Y3,_,_) :-
        within_tolerance(X1,Y1,X2,Y2),
        within_tolerance(X1,Y1,X3,Y3),
        within_tolerance(X2,Y2,X3,Y3), !.
match_3_pts(X1,Y1,_,_,X2,Y2,_, ,_,_,X3,Y3) :-
        within_tolerance(X1,Y1,X2,Y2),
        within_tolerance(X1,Y1,X3,Y3),
        within_tolerance(X2,Y2,X3,Y3), !.
match_3_pts(_,_,X1,Y1,X2,Y2,_,_,_,_,X3,Y3) :-
```

```
        within_tolerance(X1,Y1,X2,Y2),
        within_tolerance(X1,Y1,X3,Y3),
        within_tolerance(X2,Y2,X3,Y3), !.
match_3_pts(X1,Y1,_,_,_,_,X2,Y2,X3,Y3,_,_) :-
        within_tolerance(X1,Y1,X2,Y2),
        within_tolerance(X1,Y1,X3,Y3),
        within_tolerance(X2,Y2,X3,Y3), !.
match_3_pts(_,_,X1,Y1,_,_,X2,Y2,X3,Y3,_,_) :-
        within_tolerance(X1,Y1,X2,Y2),
        within_tolerance(X1,Y1,X3,Y3),
        within_tolerance(X2,Y2,X3,Y3), !.
match_3_pts(X1,Y1,_,_,_,_,X2,Y2,_,_,X3,Y3) :-
        within_tolerance(X1,Y1,X2,Y2),
        within_tolerance(X1,Y1,X3,Y3),
        within_tolerance(X2,Y2,X3,Y3), !.
match_3_pts(_,_,X1,Y1,_,_,X2,Y2,_,_,X3,Y3) :-
        within_tolerance(X1,Y1,X2,Y2),
        within_tolerance(X1,Y1,X3,Y3),
        within_tolerance(X2,Y2,X3,Y3), !.

/*************************************************************************/
/*********  "elim_incomplete_trees" subordinate predicates  *********/
/*************************************************************************/

/* Succeeds if top node of tree is anchored to an edge by means of */
/* an edge-intersection.  If so, the tree is 'complete', since     */
/* each leaf node is anchored by means of a region vertex.         */
complete_tree(tree([I,J],L,R)) :-
        ordered(I,J,I1,J1),
        tbdry(_,_,[I1,J1],[X,Y|B],LPt),
        edge_bdry_intersection(_,[I1,J1],[X,Y]), !.
complete_tree(tree([I,J],L,R)) :-
        ordered(I,J,I1,J1),
        tbdry(_,_,[I1,J1],[X,Y|B],LPt),
        edge_bdry_intersection(_,[I1,J1],LPt), !.

/* Retracts all bdrys associated with nodes in 'tree' */
eliminate_tree_tbdrys([]).
eliminate_tree_tbdrys(tree([I,J],L,R)) :-
        retract_succeed(tree([I,J],_,_)),
        ordered(I,J,I1,J1),
        retract_succeed(tbdry(_,_,[I1,J1],_,_)),
        eliminate_tree_tbdrys(L),
        eliminate_tree_tbdrys(R), !.


/*************************************************************************/
/*********** "find_exact_opp_pt" subordinate predicates  ************/
/*************************************************************************/

update_opp_edge(I,J,[LX,LY,c(C),X2,Y2|OP]) :-
        opposite_edge(OE),
        update_opp_edge2(I,J,[LX,LY,c(C),X2,Y2|OP],OE), !.

update_opp_edge2(I,J,[LX,LY,c(C),X2,Y2|OP],[Xa,Ya,H,Xb,Yb|OE]) :-
        on_line(LX,LY,Xa,Ya,Xb,Yb),
        retract(opposite_edge(OE0)),
        update_opp_edge3([LX,LY,X2,Y2],[Xa,Ya],OE0,OE1),
        assert(opposite_edge(OE1)), !.
update_opp_edge2(I,J,[LX,LY,c(C),X2,Y2|OP],[Xa,Ya,H,Xb,Yb|OE]) :-
        update_opp_edge2(I,J,[LX,LY,c(C),X2,Y2|OP],[Xb,Yb|OE]), !.
update_opp_edge2(I,J,[LX,LY,c(C),X2,Y2|OP],[]).

/* If OP is counterclockwise along opp edge: */
```

```
update_opp_edge3([LX,LY,X2,Y2],[Xa,Ya],[Xa,Ya,H,Xb,Yb|OE0],
        [Xa,Ya,h(o),X2,Y2,h(ccw),LX,LY,H,Xb,Yb|OE0]) :-
        on_ray(X2,Y2,LX,LY,Xa,Ya),!.
/* If OP is clockwise along opp edge: */
update_opp_edge3([LX,LY,X2,Y2],[Xa,Ya],[Xa,Ya,H,Xb,Yb|OE0],
        [Xa,Ya,H,L2,L2,h(cw),X2,Y2,h(o),Xb,Yb|OE0]) :-
        on_ray(X2,Y2,LX,LY,Xb,Yb),!.
/* If OP is neither clockwise or ccw then it goes into HCA interior or it */
/* goes toward goal into HCA exterior, so it says nothing about opp point */
update_opp_edge3([LX,LY,X2,Y2],[Xa,Ya],[Xa,Ya,H,Xb,Yb|OE0],
        [Xa,Ya,H,Xb,Yb|OE0]) :- !.
/* If LX,LY is not on current opp edge segment, recurse to next segment */
update_opp_edge3([LX,LY,X2,Y2],[Xa,Ya],[Xb,Yb,H|OE0],[Xb,Yb,H|OE1]) :-
        not(same([Xa,Ya],[Xb,Yb])),
        update_opp_edge3([LX,LY,X2,Y2],[Xa,Ya],OE0,OE1), !.


/*************************************************************************/
/***********  "update_opposite_edge" subordinate predicates  ********/
/*************************************************************************/

/* new_opp_pt finds a new (& correct) opposite point if one is   */
/* present, or rtns orig opp pt. If it recurses thru whole list  */
/* with h(o) for each edge, initial opp pt was good.  If it      */
/* finds a label other than h(o), then it needs new opp point.   */
new_opp_pt(_,[_,_,h(o),_,_],[Xopp,Yopp]) :-
        opposite_point(Xopp,Yopp), !.
new_opp_pt(_,[X1,Y1,h(o),X2,Y2|OE],P) :-
        new_opp_pt([X1,Y1],[X2,Y2|OE],P), !.
new_opp_pt(_,[X1,Y1,h(ccw),X2,Y2,_,X3,Y3],P) :-
        optimal_path([X1,Y1|_],Cccw),
        oe_bisection_search(X2,Y2,X3,Y3,Xcw,Ycw),
        optimal_path([Xcw,Ycw|_],Ccw),
        calc_opp_pt(X1,Y1,Xcw,Ycw,Cccw,Ccw,P), !.
new_opp_pt([X0,Y0],[X1,Y1,h(cw),X2,Y2|_],P) :-
        optimal_path([X2,Y2|_],Ccw),
        oe_bisection_search(X1,Y1,X0,Y0,Xccw,Yccw),
        optimal_path([Xccw,Yccw|_],Cccw),
        calc_opp_pt(X2,Y2,Xccw,Yccw,Ccw,Cccw,P), !.

oe_bisection_search(X1,Y1,X2,Y2,X,Y) :-          /* Bisect edge; compute OP */
        Xi0 is X1 + (X2-X1)/2,                   /* from midpt; if OP starts*/
        Yi0 is Y1 + (Y2-Y1)/2,                   /* along edge, this is the */
        round_to_4decpl(Xi0,Xi),                 /* point we are looking for*/
        round_to_4decpl(Yi0,Yi),
        optimal_path([Xi,Yi,c(C),X,Y|OP]),
        on_line(X,Y,X1,Y1,X2,Y2),!.
oe_bisection_search(X1,Y1,X2,Y2,X,Y) :-          /* If OP does not start on */
        Xi0 is X1 + (X2-X1)/2,                   /* edge, search between    */
        Yi0 is Y1 + (Y2-Y1)/2,                   /* point 1 and point i.    */
        round_to_4decpl(Xi0,Xi),
        round_to_4decpl(Yi0,Yi),
        not(within_tolerance(Xi,Yi,X1,Y1)),
        oe_bisection_search(X1,Y1,Xi,Yi,X,Y), !.
oe_bisection_search(X1,Y1,X2,Y2,X,Y) :-          /* If no correct OP is     */
        Xi is X1 + (X2-X1)/2,                    /* found, return Pt 1 as   */
        Yi is Y1 + (Y2-Y1)/2,                    /* answer. This case       */
        within_tolerance(Xi,Yi,X1,Y1),           /* SHOULD NOT HAPPEN, so   */
        tell(missing_ops),                       /* print a warning.        */
        write('Need OP for start-point '),
        write([Xi,Yi]),nl,
        tell(user),
        write('WARNING: missing optimal path from opposite edge'),nl,
        write('         for predicate ''oe_bisection_search'''),nl, !.
```

```
/*************************************************************************/
/*****************  "output" subordinate predicates  *****************/
/*************************************************************************/

write_bdrys_to_file(hca_opm,[]).
write_bdrys_to_file(hca_opm,[[[I,J],B,LPt]|BdrySet]) :-
        tell(hca_opm),
        write('bdry('),
        write([I,J]),write(','),write(B),
        write(').'),nl,
        write_bdrys_to_file(hca_opm,BdrySet).

write_iter_to_screen :-
        retract(number_of_iter(I)),
        Iplus1 is 1 + 1,
        assert(number_of_iter(Iplus1)),
        tell(user),
        write('consistency check '),
        write(I),nl, !.
write_heading :-
        region_vertices([X,Y|R]), goal_point(Xg,Yg),
        cons([X,Y|R],[X,Y],Region),
        tell(hca_opm),
        listing(title),
        write('region('),write(Region),write(').'),nl,
        write('goal(['),write(Xg),write(','),write(Yg),write(']).'),nl,
        listing(tree), !.

write_to_screen(X) :-
        tell(user),
        write(X),nl,!.




/*************************************************************************/
/***********************  utility predicates  ************************/
/*************************************************************************/

/* Succeeds if line segments intersect, but do not share an endpoint. */
interior_intersection([X1,Y1|B1],LPt1,[X2,Y2|B2],LPt2,IntPt,B1tr,B2tr) :-
        not(same(LPt1,LPt2)),            /* If inters at endpt, fails; if not,*/
        not(same([X1,Y1],[X2,Y2])),      /* and intersects somewhere, succeeds*/
        not(same([X1,Y1],LPt2)),             /* Assumes B1 & B2 intersect     */
        not(same([X2,Y2],LPt1)),             /* in at most one point.         */
        bdry_intersection([X1,Y1|B1],[X2,Y2|B2],IntPt,B1tr,B2tr), !.
interior_intersection([X1,Y1|B1],[],[X2,Y2|B2],[],IntPt,B1tr,B2tr) :-  /* Full*/
        bdry_intersection([X1,Y1|B1],[X2,Y2|B2],IntPt,B1tr,B2tr), !. /*bdrys*/
                                                        /* Full bdrys*/
/*
 * 'bdry_intersection' determines the intersection of two boundaries,
 * or fails if there is no intersection.  The boundaries are
 * piecewise linear,and are represented as a list of points,
 * ie, [x1,y1,x2,y2,x3,y3,...].  Tolerance is allowed.
 * Once an intersection is found, it is cached to speed up future references */
bdry_intersection(B1,B2,[Xi2,Yi2],B1tr2,B2tr2) :-
        bdry_intersection1(appx,B1,B2,[Xi,Yi],B1trunc,B2trunc),
        Xi2 is (floor(Xi*10000)/10000),
        Yi2 is (floor(Yi*10000)/10000),
        replace_last_coords(B1trunc,[Xi2,Yi2],B1tr2),
        replace_last_coords(B2trunc,[Xi2,Yi2],B2tr2),
        asserta(bdry_intersection(B1,B2,[Xi2,Yi2],B1tr2,B2tr2)), !.
```

```
/* 'bdry_intersection_exact' is like 'bdry_intersection', except that
 * no tolerance is allowed on intersection point being interior to
 * both bdrys, and bdry_intersections are not cached. */
bdry_intersection_exact(B1,B2,[Xi2,Yi2],B1tr2,B2tr2) :-
        bdry_intersection1(exact,B1,B2,[Xi,Yi],B1trunc,B2trunc),
        Xi2 is (floor(Xi*10000)/10000),
        Yi2 is (floor(Yi*10000)/10000),
        replace_last_coords(B1trunc,[Xi2,Yi2],B1tr2),
        replace_last_coords(B2trunc,[Xi2,Yi2],B2tr2),!.
/* Check if any segment of bdry 1 matches the 1st segment of bdry 2. */
bdry_intersection1(Prec,[X11,Y11,X12,Y12|B1],[X21,Y21,X22,Y22|B2],[Xi,Yi],
        B1trunc,B2trunc) :-
        bdry_intersection2(Prec,[X11,Y11,X12,Y12|B1],
        [X21,Y21,X22,Y22],[Xi,Yi],B1trunc,B2trunc), !.
/* Recursively check the next segment of boundary 2 with all of bdry 1.  */
bdry_intersection1(Prec,[X11,Y11,X12,Y12|B1],[X21,Y21,X22,Y22|B2],[Xi,Yi],
        B1trunc,[X21,Y21|B2trunc]) :-
        bdry_intersection1(Prec,[X11,Y11,X12,Y12|B1],
        [X22,Y22|B2],[Xi,Yi],B1trunc,B2trunc), !.
/* Recursively see if any seg of bdry 1 matches the 1st segment of bdry 2. */
bdry_intersection2(appx,[X11,Y11,X12,Y12|B1],
        [X21,Y21,X22,Y22],[Xi,Yi],[X11,Y11,Xi,Yi],[X21,Y21,Xi,Yi]) :-
        line_intersection(X11,Y11,X12,Y12,X21,Y21,X22,Y22,Xi,Yi),
        between(Xi,X11,X12),
        between(Yi,Y11,Y12),                /* Check if pt i is between   */
        between(Xi,X21,X22),                /* endpoints of both segments */
        between(Yi,Y21,Y22), !.             /* inclusively                */
bdry_intersection2(appx,[X11,Y11,X12,Y12|B1],
        [X21,Y21,X22,Y22],[Xi,Yi],[X11,Y11|B1trunc],B2trunc) :-
        bdry_intersection2(appx,[X12,Y12|B1],
        [X21,Y21,X22,Y22],[Xi,Yi],B1trunc,B2trunc), !.
bdry_intersection2(exact,[X11,Y11,X12,Y12|B1],
        [X21,Y21,X22,Y22],[Xi,Yi],[X11,Y11,Xi,Yi],[X21,Y21,Xi,Yi]) :-
        line_intersection(X11,Y11,X12,Y12,X21,Y21,X22,Y22,Xi,Yi),
        exact_between(Xi,X11,X12),
        exact_between(Yi,Y11,Y12),           /* Check if pt i is between   */
        exact_between(Xi,X21,X22),           /* endpoints of both segments */
        exact_between(Yi,Y21,Y22), !.        /* inclusively                */
bdry_intersection2(exact,[X11,Y11,X12,Y12|B1],
        [X21,Y21,X22,Y22],[Xi,Yi],[X11,Y11|B1trunc],B2trunc) :-
        bdry_intersection2(exact,[X12,Y12|B1],
        [X21,Y21,X22,Y22],[Xi,Yi],B1trunc,B2trunc), !.

replace_last_coords([_,_],[X,Y],[X,Y]).
replace_last_coords([Xi,Yi|L],[X,Y],[Xi,Yi|L2]) :-
        replace_last_coords(L,[X,Y],L2), !.

get_tbdryIJorJI(F,C,[I,J],B,LPt) :- tbdry(F,C,[I,J],B,LPt).
get_tbdryIJorJI(F,C,[I,J],B,LPt) :- tbdry(F,C,[J,I],B,LPt).

reassert_tbdrys(F,[],Ctr).
reassert_tbdrys(F,[[[I,J],B,LPt]|A3],Ctr) :-
        assert(tbdry(F,Ctr,[I,J],B,LPt)),
        C2 is Ctr+1,
        reassert_tbdrys(F,A3,C2), !.

order_indices([],[]) :- !.
order_indices([[[I,J],B,LPt]|Rest],[[[I1,J1],B,LPt]|RevisedRest]) :-
        ordered(I,J,I1,J1),
        order_indices(Rest,RevisedRest), !.

adjacent_bdrys(I,J,J,K,I,J,J,K) :- !.
adjacent_bdrys(I,J,K,J,I,J,J,K) :- !.
adjacent_bdrys(I,J,I,K,J,I,I,K) :- !.
```

```
adjacent_bdrys(I,J,K,I,J,I,I,K) :- !.

adjacent_edges(I,J) :- J is I + 1.
adjacent_edges(1,N) :- number_of_edges(N).

get_initbdryIJorJI(I,J,B) :- initbdry([I,J],B).
get_initbdryIJorJI(I,J,B) :- initbdry([J,I],B).

get_tbdrys(_) :-
        assert(bdry_list([])),
        tbdry(_,_,[I,J],B,LPtB),
        retract_cut(bdry_list(L)),
        assert(bdry_list([[[I,J],B,LPtB]|L])),
        fail.
get_tbdrys(A) :- retract(bdry_list(A)), !.

reset_last_pts :-                          /* Insures that intersecting */
        tbdry(F1,C1,[I,J],B1,[LX1,LY1]),   /* bdrys have identical last points */
        tbdry(F2,C2,[K,L],B2,[LX2,LY2]),
        adjacent_bdrys(I,J,K,L,I1,J1,K1,L1),
        not(same([I,J],[K,L])),
        within_tolerance(LX1,LY1,LX2,LY2),
        retract_cut(tbdry(F2,C2,[K,L],B2,[LX2,LY2])),
        asserta(tbdry(F2,C2,[K,L],B2,[LX1,LY1])),
        fail, !.
reset_last_pts :- !.

edge_adjacent_to_bdry(I,I,J).
edge_adjacent_to_bdry(J,I,J).

bdry_starts_at_edge(I,J,J) :- Diff is I-J, abs(Diff,1), !.
bdry_starts_at_edge(I,J,1) :- Diff is I-J, abs(Diff,1), !.
bdry_starts_at_edge(1,N,1) :- number_of_edges(N), !.
bdry_starts_at_edge(1,N,N) :- number_of_edges(N), !.
bdry_starts_at_edge(N,1,1) :- number_of_edges(N), !.
bdry_starts_at_edge(N,1,N) :- number_of_edges(N), !.

/* Opp Pt is located proportional to OP costs at each end */
calc_opp_pt(X1,Y1,X2,Y2,C1,C2,[Xopp,Yopp]) :-
        distance(X1,Y1,X2,Y2,D12),
        DelX is X2 - X1, DelY is Y2 - Y1,
        Xopp0 is X1 + ((D12+C2-C1)/D12)*(DelX/2),
        Yopp0 is Y1 + ((D12+C2-C1)/D12)*(DelY/2),
        round_to_4decpl(Xopp0,Xopp),
        round_to_4decpl(Yopp0,Yopp), !.

/* Rounds off a number to four decimal places (to allow unification  */
/* with manually input optimal paths) */
round_to_4decpl(X,Xr) :- Xr is (floor((X+0.00005)*10000))/10000.
```

```
/*****************************************************************************
 *****************************************************************************
 *
 * "bgutils" contains supporting predicates used by the "bg" files.
 *
 * Consulted and called by "bg".
 *
 * Updated 12 Jan 89.
 *
 *****************************************************************************
 *****************************************************************************/

/* CONSTANTS: Dimensions of input map */
minX(0).
maxX(80).
minY(0).
maxY(100).

pi(3.14159).

/* "precision" is the max number of line segments to compute */
/* for each 1vis boundary, and twice the number for 2vis boundaries.  */
precision(20).

tolerance(0.05).    /* Pts closer than this are usually not distinguished */
/* tolerance(0.015). */

/*****************************************************************************/
/******************  general utility predicates  *********************/
/*****************************************************************************/

between(B,A,C) :-
        tolerance(T),
        Bplus is B + T,
        Cplus is C + T,
        A =< Bplus,
        B =< Cplus, !.
between(B,A,C) :-
        tolerance(T),
        Bminus is B - T,
        Cminus is C - T,
        A >= Bminus,
        B >= Cminus, !.

exact_between(B,A,C) :-
        A =< B,
        B =< C, !.
exact_between(B,A,C) :-
        A >= B,
        B >= C, !.

strictly_between(B,A,C) :-
        A < B,
        B < C, !.
strictly_between(B,A,C) :-
        A > B,
        B > C, !.

get_counter_and_increment(Ctr) :-
        retract(ctr(Ctr)),
        Cplus1 is Ctr + 1,
        assert(ctr(Cplus1)), !.
```

```
ordered(I,J,I,J) :- I =< J, !.
ordered(I,J,J,I) :- !.

abs(X,X) :- X >= 0.
abs(X,Y) :- X =< 0, Y is -X.

retract_succeed(P) :- retract(P), !.  /* Always succeeds, fails on backtrack */
retract_succeed(_) :- !.

retract_cut(P) :- retract(P),!.       /* Retracts P, fails on backtracking */

unify_cut(P) :- call(P),!.  /* Gets 1st instance of P, fails on backtracking */

/* fails the first time called, then succeeds, toggling thereafter */
fail_succeed :- not(failed), assert(failed), !, fail, !.
fail_succeed :- failed, retract(failed), !.


get_last_pt([Xlast,Ylast],Xlast,Ylast).
get_last_pt([X,Y|Rest],Xlast,Ylast) :-
        get_last_pt(Rest,Xlast,Ylast).

get_last_list([Last],Last).
get_last_list([F|Rest],Last) :-
        get_last_list(Rest,Last).

within_tolerance(X1,Y1,X2,Y2) :-
        tolerance(Tolerance),
        DelX is X1-X2,
        DelX < Tolerance,
        DelX > -Tolerance,
        DelY is Y1-Y2,
        DelY < Tolerance,
        DelY > -Tolerance, !.

same_set(Set1,Set2) :-
        same(Set1,Set2), !.
same_set([A|Set1],Set2) :-
        match_and_delete(A,Set2,Set2LessA),
        same_set(Set1,Set2LessA), !.

match_and_delete(A,[A],[]).
match_and_delete(A,[A|Rest],Rest).
match_and_delete(A,[B|Set],[B|SetLessA]) :-
        match_and_delete(A,Set,SetLessA).

/* input: starting and ending integers */
/* output: list of lists of the form [[1,2],[2,3],...,[N-1,N],[N,1]] */
/* where an index pair appears the num of times its initbdry appears */
index_list_ItoJ(J,J,[[J,1]]) :-  !.
index_list_ItoJ(I,J,[[I,Iplus1]|Rest]) :-
        Iplus1 is I + 1,
        index_list_ItoJ(Iplus1,J,Rest), !.

/* returns the number of initbdry([I,J],_)'s asserted */
number_of_IJ_initbdrys(I,J,N) :-
        assert(temp_num(0)),
        initbdry([I,J],_),
        retract_cut(temp_num(K)),
        Kplus1 is K+1,
        assert(temp_num(Kplus1)),
        fail, !.
number_of_IJ_initbdrys(I,J,N) :-
        retract_cut(temp_num(N)), !.
```

```
/* input: list of 3-element bdry lists of form [[[1,2],B1,LPt1],...] */
/* output: list of lists of the form [[1,2],[2,3],...] where each */
/*         bdry in input list is represented by its index list    */
index_list([],[]) :- !.
index_list([[[I,J],_,_]|Rest],[[I,J]|RevList]) :-
        index_list(Rest,RevList), !.

/* input: Num of edges and list of indices of each bdry previously asserted */
/* output: list of indices of each bdry not previously asserted */
complement_index_list(N,[[1,2]|InList],OutList) :-   /* If [1,2] is first */
        complement_index_list1(2,N,InList,[],OutList), !.
complement_index_list(N,InList,OutList) :-           /* If [1,2] is not first */
        complement_index_list1(2,N,InList,[[1,2]],OutList), !.
/* If [1,2] is first OR last, do not include it in complement list */
/* If [1,2] is neither first NOR last, include it */
complement_index_list1(N,N,[],First,[[N,1]|First]) :-  !.
complement_index_list1(N,N,[[N,1]],First,First) :-  !.
complement_index_list1(N,N,[[1,2]],First,[[N,1]]) :-  !.
complement_index_list1(N,N,[[N,1],[1,2]],First,[]) :-  !.
complement_index_list1(I,N,[[I,J]|Rest],First,RevIndexList) :-
        Iplus1 is I + 1,
        complement_index_list1(Iplus1,N,Rest,First,RevIndexList), !.
complement_index_list1(I,N,[[J,K]|Rest],First,[[I,Iplus1]|RevIndexList]) :-
        Iplus1 is I + 1,
        not(same(I,J)),
        complement_index_list1(Iplus1,N,[[J,K]|Rest],First,RevIndexList), !.

set_subtraction(L,[],L) :- !.
set_subtraction(L1,[A|L2],L4) :-
        delete_from_list(A,L1,L3),
        set_subtraction(L3,L2,L4), !.

delete_from_list(A,[],[]) :- !.
delete_from_list(A,[A|L],L2) :-
        delete_from_list(A,L,L2), !.
delete_from_list(A,[B|L],[B|L2]) :-
        delete_from_list(A,L,L2), !.

/*
 * predicates related to rotation and translation of the boundary.
 */

compute_angle_of_rotation(Xo,Yo,X,Y,Angle) :-     /* Computes angle to rot.*/
        DelX is X-Xo,                             /* the x-axis to the     */
        DelX >= 0,                                /* vector (Xo,Yo)->(X,Y) */
        DelY is Y-Yo,                             /* when the angle is     */
        Angle is -asin(DelY/sqrt(DelX^2+DelY^2)). /* between -pi/2 & pi/2, */
compute_angle_of_rotation(Xo,Yo,X,Y,Angle) :-
        DelY is Y-Yo,                             /* ...when the angle is  */
        DelY >= 0,                                /* between pi/2 & pi.    */
        DelX is X-Xo,
        pi(Pi),
        Angle is -Pi+asin(DelY/sqrt(DelX^2+DelY^2)).
compute_angle_of_rotation(Xo,Yo,X,Y,Angle) :-
        DelY is Y-Yo,                             /* ...when the angle is  */
        DelY < 0,                                 /* between pi & 3pi/2.   */
        DelX is X-Xo,
        pi(Pi),
        Angle is Pi+asin(DelY/sqrt(DelX^2+DelY^2)).

invert_bdry :-                                    /* Reflects the boundary  */
        retract(bdry(P)),                         /* about the vertical line */
```

```
        invert_bdry(R,Vx,R_inv),                /* X=Vx, where Vx is the   */
        assert(bdry(R_inv)), !.                 /* bdry vertex or last pair*/
invert_bdry([X,Y],X,[X,Y]).                     /* of coords in bdry list  */
invert_bdry([X,Y|R],Xrefl,[Xinv,Yinv|Rinv]) :-
        invert_bdry(R,Xrefl,Rinv),
        invert_coords(X,Y,Xrefl,Xinv,Yinv).
invert_coords(X,Y,Xrefl,Xinv,Y) :-
        Xinv is 2*Xrefl - X.

rotate_bdry(Angle) :-
        retract(bdry(L)),
        rotate_bdry(L,Angle,_,_,Lrot),
        assert(bdry(Lrot)), !.
rotate_bdry([X,Y],Angle,X,Y,[X,Y]).             /* Assumes last item in */
                                                /*list is bdry vertex */
rotate_bdry([X,Y|L],Angle,Xo,Yo,[Xrot,Yrot|Lrot]) :-
        rotate_bdry(L,Angle,Xo,Yo,Lrot),
        translate_point(X,Y,Xo,Yo,Xtr,Ytr),
        rotate_point(Xtr,Ytr,Angle,Xrot1,Yrot1),
        re_translate_point(Xrot1,Yrot1,Xo,Yo,Xrot,Yrot).
rotate_point(X,Y,Angle,Xrot,Yrot) :-
        Xrot is X*cos(Angle) + Y*sin(Angle),
        Yrot is Y*cos(Angle) - X*sin(Angle).

rotate2_bdry(Angle) :-
        retract(bdry(L)),
        rotate2_bdry(L,Angle,_,_,Lrot),
        assert(bdry(Lrot)), !.
rotate2_bdry([X,Y],Angle,X,Y,[]).               /* Assumes last item in */
                        /*list is origin of rotation, but not on boundary */
rotate2_bdry([X,Y|L],Angle,Xo,Yo,[Xrot,Yrot|Lrot]) :-
        rotate2_bdry(L,Angle,Xo,Yo,Lrot),
        translate_point(X,Y,Xo,Yo,Xtr,Ytr),
        rotate_point(Xtr,Ytr,Angle,Xrot1,Yrot1),
        re_translate_point(Xrot1,Yrot1,Xo,Yo,Xrot,Yrot).

translate_point(X,Y,Xo,Yo,Xtr,Ytr) :-
        Xtr is X-Xo,
        Ytr is Y-Yo.
re_translate_point(Xtr,Ytr,Xo,Yo,X,Y) :-
        X is Xtr+Xo,
        Y is Ytr+Yo.



/*
 *  Miscellaneous utility predicates.
 */

debug_list(Filename) :-
        tell(Filename),
        listing(region_vertices),listing(title),listing(goal_point),
        listing(tbdry),listing(initbdry),listing(tree).

reverse_list([],[]).
reverse_list([X|L],RevLconsX) :-
        reverse_list(L,RevL),
        cons(RevL,[X],RevLconsX).

reverse_edge_list([X,Y],[X,Y]).
reverse_edge_list([X,Y,H|L],RevLconsX) :-
        reverse_edge_list(L,RevL),
        cons(RevL,[H,X,Y],RevLconsX).
```

```
reverse_path_list([X,Y],[X,Y]) :- !.
reverse_path_list([X1,Y1,X2,Y2],[X2,Y2,X1,Y1]) :- !.
reverse_path_list([X1,Y1,X2,Y2,X3,Y3],[X3,Y3,X2,Y2,X1,Y1]) :- !.
reverse_path_list([X1,Y1,X2,Y2,X3,Y3,X4,Y4],[X4,Y4,X3,Y3,X2,Y2,X1,Y1]) :- !.
reverse_path_list([X1,Y1,X2,Y2,X3,Y3,X4,Y4,X5,Y5],
        [X5,Y5,X4,Y4,X3,Y3,X2,Y2,X1,Y1]) :- !.
reverse_path_list([X1,Y1,X2,Y2,X3,Y3,X4,Y4,X5,Y5,X6,Y6],
        [X6,Y6,X5,Y5,X4,Y4,X3,Y3,X2,Y2,X1,Y1]) :- !.
reverse_path_list([X1,Y1,X2,Y2,X3,Y3,X4,Y4,X5,Y5,X6,Y6,X7,Y7],
        [X7,Y7,X6,Y6,X5,Y5,X4,Y4,X3,Y3,X2,Y2,X1,Y1]) :- !.
reverse_path_list([X1,Y1,X2,Y2,X3,Y3,X4,Y4,X5,Y5,X6,Y6,X7,Y7,X8,Y8],
        [X8,Y8,X7,Y7,X6,Y6,X5,Y5,X4,Y4,X3,Y3,X2,Y2,X1,Y1]) :- !.
reverse_path_list([X1,Y1,X2,Y2,X3,Y3,X4,Y4,X5,Y5,X6,Y6,X7,Y7,X8,Y8,X9,Y9],
        [X9,Y9,X8,Y8,X7,Y7,X6,Y6,X5,Y5,X4,Y4,X3,Y3,X2,Y2,X1,Y1]) :- !.
reverse_path_list([X,Y|L],RevLconsXY) :-
        reverse_path_list(L,RevL),
        cons(RevL,[X,Y],RevLconsXY), !.

/* Return the point of intersection of two lines, fail if parallel. */
/* Note: next 4 rules are included to retain precision where possible. */
/* If lines share a point, that point is the intersection: */
line_intersection(X1,Y1,X2,Y2,X1,Y1,X4,Y4,X1,Y1) :- !.
line_intersection(X1,Y1,X2,Y2,X2,Y2,X4,Y4,X2,Y2) :- !.
line_intersection(X1,Y1,X2,Y2,X3,Y3,X1,Y1,X1,Y1) :- !.
line_intersection(X1,Y1,X2,Y2,X3,Y3,X2,Y2,X2,Y2) :- !.
line_intersection(X1,Y1,X2,Y2,X3,Y3,X4,Y4,Xi,Yi) :-
        not(X2==X1),                    /* Handle separately if */
        not(X4==X3),                    /*    one line is vertical */
        Ma is (Y2-Y1)/(X2-X1),          /* Slope of 1st line */
        Ba is Y1-Ma*X1,                 /* Y-intecept of 1st line */
        Mb is (Y4-Y3)/(X4-X3),          /* Slope of 2nd line */
        Bb is Y3-Mb*X3,                 /* Y-intecept of 2nd line */
        not(Mb==Ma),                    /* This happens if lines are parallel */
        Xi is (Ba-Bb)/(Mb-Ma),
        Yi is Mb*Xi + Bb, !.
line_intersection(X1,Y1,X1,Y2,X3,Y3,X4,Y4,Xi,Yi) :-
        not(X4==X3),                    /* Case where 1st line is vertical */
        Mb is (Y4-Y3)/(X4-X3),          /* Fails if both lines vertical */
        Bb is Y3-Mb*X3,
        Xi is X1,
        Yi is Mb*Xi + Bb, !.
line_intersection(X1,Y1,X2,Y2,X3,Y3,X3,Y4,Xi,Yi) :-
        not(X2==X1),                    /* Case where 2nd line is vertical */
        Ma is (Y2-Y1)/(X2-X1),          /* Fails if both lines vertical */
        Ba is Y1-Ma*X1,
        Xi is X3,
        Yi is Ma*Xi + Ba, !.
line_intersection(X1,Y1,X2,Y2,X1,Y1,X2,Y2,X1,Y1). /* If lines are identical */
                                /* return the 1st vertex as intersection point.*/
line_intersection(X1,Y1,X1,Y2,X1,Y3,X1,Y4,X1,Y1).    /* Case where lines are */
        /* vertical & coincident;  return 1st vertex of 1st line as int pt.  */
line_intersection(X1,Y1,X1,Y2,X3,Y3,X3,Y4,Xi,Yi) :-  /* Case where lines are */
        !, fail.                    /* both vertical, but not coincident; fail. */
line_intersection(X1,Y1,X2,Y2,X3,Y3,X4,Y4,X1,Y1) :-  /* Coincident lines */
        not(X2==X1),
        not(X4==X3),
        Ma is (Y2-Y1)/(X2-X1),          /* Slope of 1st line */
        Ba is Y1-Ma*X1,                 /* Y-intecept of 1st line */
        Mb is (Y4-Y3)/(X4-X3),          /* Slope of 2nd line */
        Bb is Y3-Mb*X3,                 /* Y-intecept of 2nd line */
        Ma==Mb,                         /* Parallel */
        Ba==Bb, !.                      /* Same y-intercepts */
         /* If lines are coincident, return 1st vertex of 1st line as int pt */
```

```
virtual_vertex(X1,Y1,X2,Y2,X3,Y3,X4,Y4,Xv,Yv) :-    /* the virtual vertex is */
        line_intersection(X1,Y1,X2,Y2,X3,Y3,X4,Y4,Xv,Yv),!.  /* the point of */
                                                  /* intersection of the lines. */

distance(X1,Y1,X2,Y2,D) :-
        D is sqrt((X2-X1)^2 + (Y2-Y1)^2).

/* Counts length of a list */
list_length([],0).
list_length([X|Rest],Iplus1) :-
        list_length(Rest,I),
        Iplus1 is I + 1.

/* Computes length of path P */
path_length([],0).
path_length([_,_],0).
path_length([X1,Y1,c(C),X2,Y2|P],D) :-      /* If cost data is present, find */
        path_length([X2,Y2|P],D1),          /* weighted cost of path;       */
        distance(X1,Y1,X2,Y2,D2),
        D is D1 + C*D2, !.
path_length([X1,Y1,X2,Y2|P],D) :-           /* If cost data is not present,  */
        path_length([X2,Y2|P],D1),          /* find Euclidean length of path */
        distance(X1,Y1,X2,Y2,D2),
        D is D1 + D2, !.

/* Computes length of edge P */
edge_length([],0).
edge_length([_,_],0).
edge_length([X1,Y1,V,X2,Y2|P],D) :-
        edge_length([X2,Y2|P],D1),
        distance(X1,Y1,X2,Y2,D2),
        D is D1 + D2, !.


/* Computes the distance between pts 1 & 2 along path P */
/* First travels down the path until pt 1 is found */
/* then constructs list btwn pt1 & 2, then finds its length */
path_distance(X1,Y1,X2,Y2,[X1,Y1|P],D) :-
        path_distance2(X1,Y1,X2,Y2,[X1,Y1|P],P12),
        path_length(P12,D), !.
path_distance(X1,Y1,X2,Y2,[X3,Y3,c(C)|P],D) :-
        path_distance(X1,Y1,X2,Y2,P,D).
path_distance2(X1,Y1,X1,Y1,_,[]).
path_distance2(X1,Y1,X2,Y2,[X2,Y2|P],[X2,Y2]).
path_distance2(X1,Y1,X2,Y2,[X3,Y3,c(C)|P],[X3,Y3,c(C)|IntList]) :-
        path_distance2(X1,Y1,X2,Y2,P,IntList), !.

same(A,A).   /* succeeds if both args are the same, fails otherwise */

/* concat arg 2 (atom), onto end of arg 1 (list), return as arg 3 (list) */
cons([],B,B) :- !.
cons([X|B1],B2,[X|B3]) :-
        cons(B1,B2,B3),!.

/* A robust arccosine routine (in C-Prolog, acos(1) bombs) */
arccos(X,0) :-
        X > 0.99999,  !.
arccos(X,A) :- A is acos(X), !.

/* Remove the last pair of coords from the 1st arg, return as 2nd arg */
remove_last_pt([X,Y,X2,Y2],[X,Y]).
remove_last_pt([X,Y|L],[X,Y|RevL]) :-
        remove_last_pt(L,RevL), !.
```

```
convert_degr_to_rads(Angle_in_Deg,Angle_in_Rad) :-
        degr_to_rads_factor(F),
        Angle_in_Rad is F*Angle_in_Deg, !.
convert_rads_to_degr(T2,T2Deg) :-
        rads_to_degr_factor(F2),
        T2Deg is F2*T2, !.
degr_to_rads_factor(F) :- F is 3.14159/180.
rads_to_degr_factor(F) :- F is 180/3.14159.

/* Succeeds if 1st path includes 2nd path, fails otherwise. */
/* Assumes both arguments are bound. */
includes_path(P,P):- !.
includes_path([X1,Y1,c(C)|P1],P2) :-
        includes_path(P1,P2), !.

/* Succeeds if arg 1 is a member of 2nd arg (a list), else fails */
member(X,[X|R]) :- !.
member(X,[X2|R]) :-
        member(X,R), !.
/* Succeeds if args 1 and 2 are members of 3rd arg (a list) */
/* in order listed, else fails */
member(X,Y,[X,Y|R]):- !.
member(X,Y,[X2,Y2|R]) :-
        member(X,Y,R), !.
/* Succeeds if args 1 thru 4 are members of 5th arg (a list) */
/* in order listed, else fails */
member(X1,Y1,X2,Y2,[X1,Y1,X2,Y2|R]):- !.
member(X1,Y1,X2,Y2,[X3,Y3,X4,Y4|R]) :-
        member(X1,Y1,X2,Y2,[X4,Y4|R]), !.

abs(A,A) :- A >= 0.
abs(A,-A) :- A < 0.

/*************************************************************************/
/* NOTE: For development purposes (until a pt-to-pt path planner is      */
/* included in the program) optimal paths from each terrain feature      */
/* vertex must be included in the mapdata file. Additionally, OP's       */
/* from each shortcutting point along the opposite edge must be incl.    */
/* There are two ways to query an optimal path:                          */
/* 1. optimal_path([X,Y|P]) will get an OP from pt X,Y if such an OP     */
/*    exists in the database.                                            */
/* 2. optimal_path([X,Y|P],C) with C unbound will get the OP from X,Y    */
/*    and determine the cost of the path.                                */
/* 3. optimal_path([X,Y|P]) will get a 'pseudo OP' from X,Y if one       */
/*    exists and there is no 'optimal_path'; this is applicable to the   */
/*    first pass only.                                                   */
/*************************************************************************/

/* Computes the cost of an optimal path, given the path in the DB       */
optimal_path(L,D) :-
        var(D),                 /* If opt path with total cost is already */
        optimal_path(L),        /* asserted, use it (bgmapdata is consulted */
        optimal_path2(L,D),!.   /* before bgutils), else compute it here. */
optimal_path2([X1,Y1,c(C),X2,Y2],D1) :-     /* If opt_path has cost data, use */
        distance(X1,Y1,X2,Y2,D), D1 is C*D, !.  /* this rule as the base case.*/
optimal_path2([X1,Y1,c(C),X2,Y2|Rest],D) :-  /* If opt path has cost data use */
        optimal_path2([X2,Y2|Rest],D2),      /* this rule as the rec. case */
        distance(X1,Y1,X2,Y2,D1), D1a is C*D1,
        D is D1a+D2, !.
optimal_path(L) :-
        pseudo_optimal_path(L).


/* succeeds if 1st pt is on line segment between pt2 & pt3 inclusive */
```

```
on_line(X1,Y1,X1,Y1,X2,Y2) :- !.
on_line(X2,Y2,X1,Y1,X2,Y2) :- !.
on_line(Xi,Yi,X1,Y1,X2,Y2) :-
        between(Xi,X1,X2), between(Yi,Y1,Y2),
        DelX is X2-X1, DelY is Y2-Y1, not(between(DelX,0,0)),
        Yj is (Xi-X2)*DelY/DelX + Y2,
        within_tolerance(Xi,Yi,Xi,Yj), !.
on_line(Xi,Yi,X1,Y1,X2,Y2) :-
        between(Xi,X1,X2), between(Yi,Y1,Y2),
        DelX is X2-X1, DelXi2 is X2-Xi,
        within_tolerance(DelXi2,Yi,DelX,Yi), !.



edge_visibility_check([Xa,Ya,Xb,Yb],[Xg,Yg], [Xa,Ya,v,Xb,Yb]) :-
        CrossProdZ is (Xb-Xa)*(Yg-Ya)-(Yb-Ya)*(Xg-Xa),
        CrossProdZ >= 0,!.  /* True if AngleGAB is between 0 and pi, */
                            /* which is true if AB is visible from G.*/
edge_visibility_check([Xa,Ya,Xb,Yb],[Xg,Yg], [Xa,Ya,h,Xb,Yb]) :- !.
edge_visibility_check([Xa,Ya,Xb,Yb|RListRest],[Xg,Yg],
        [Xa,Ya,v|RevisedRListRest]) :-
        CrossProdZ is (Xb-Xa)*(Yg-Ya)-(Yb-Ya)*(Xg-Xa),
        CrossProdZ >= 0,    /* True if AngleGAB is between 0 and pi, */
                            /* which is true if AB is visible from G.*/
        edge_visibility_check([Xb,Yb|RListRest],[Xg,Yg],RevisedRListRest),!.
edge_visibility_check([Xa,Ya,Xb,Yb|RListRest],[Xg,Yg],
        [Xa,Ya,h|RevisedRListRest]) :-
        edge_visibility_check([Xb,Yb|RListRest],[Xg,Yg],RevisedRListRest),!.

set_done_flag(Xbdry,Ybdry) :-               /* Compute bdry until it is */
        maxX(MaxX),minX(MinX),                    /* off the page by 1/2 the  */
        maxY(MaxY),minY(MinY),                    /* width of the page, to    */
        Xbdry > MinX-(MaxX-MinX)/2,               /* account for rotation.    */
        Xbdry < MaxX+(MaxX-MinX)/2,
        Ybdry > MinY-(MaxY-MinY)/2,
        Ybdry < MaxY+(MaxY-MinY)/2, !.
set_done_flag(Xbdry,Ybdry) :-      /* if bdry is off the output page, */
        assert(done),!.                  /* set "done" */

store_2vis_results(T1,T2,Y1,Y2,B,Dg,Vx,Vy) :-
        Xg is Vx + Dg*cos(B),
        Xbaseline is Xg - Y1*sin(T1),
        Xbdry is Xbaseline - Y2*sin(T2),
        Ybdry is Vy + Y2*cos(T2),
        set_done_flag(Xbdry,Ybdry),
        retract(bdry(BList)),
        assert(bdry([Xbdry,Ybdry|BList])), !.


/*
 *  Output predicates.
 */

output_init_bdrys :-
        tell(bdry_out),
        number_of_edges(N),
        write_to_bdry_file(number_of_edges,N),
        nl,
        initbdry([N1,N2],B),
        write_to_bdry_file(bdry,B,N1,N2),
        fail.
output_init_bdrys.
```

```
output_active_bdrys :-
        tell(bdry_out),nl,
        activebdry([N1,N2],B),
        write_to_bdry_file(bdry,B,N1,N2),
        fail.
output_active_bdrys.


/*
 *  Output prolog facts to file "bdry_out"
 */
write_to_bdry_file(title,Title) :-
        write_flag(no_write), !.
write_to_bdry_file(title,Title) :-
        write_flag(write),
        tell(bdry_out),
        write('title('''), write(Title), write(''').'), nl, !.

write_to_bdry_file(goal,[X,Y]) :-
        write_flag(no_write), !.
write_to_bdry_file(goal,[X,Y]) :-
        write_flag(write),
        tell(bdry_out),
        write('goal('),
        write([X,Y]),
        write(').'), nl.

write_to_bdry_file(number_of_edges,N) :-
        write_flag(no_write), !.
write_to_bdry_file(number_of_edges,N) :-
        write_flag(write),
        tell(bdry_out),
        write('number_of_edges('),
        write(N),
        write(').'), nl.

write_to_bdry_file(opposite_point,[X,Y,Xm,Ym,Xp,Yp]) :-
        write_flag(no_write), !.
write_to_bdry_file(opposite_point,[X,Y,Xm,Ym,Xp,Yp]) :-
        write_flag(write),
        tell(bdry_out),
        write('opposite_point('),
        write(X), write(','), write(Y),
        write(').'), nl,
        write('opposite_point_minus('),
        write(Xm), write(','), write(Ym),
        write(').'), nl,
        write('opposite_point_plus('),
        write(Xp), write(','), write(Yp),
        write(').'), nl.

write_to_bdry_file(region,R) :-
        write_flag(no_write), !.
write_to_bdry_file(region,R) :-
        write_flag(write),
        tell(bdry_out),
        write('region('), write(R), write(').'), nl.

write_to_bdry_file(region_elist,R) :-
        write_flag(no_write), !.
write_to_bdry_file(region_elist,R) :-
        write_flag(write),
        tell(bdry_out),
        write('region_elist('), write(R), write(').'), nl.
```

```
write_to_bdry_file(bdry,B,N1,N2) :-
        write_flag(no_write), !.
write_to_bdry_file(bdry,B,N1,N2) :-
        write_flag(write),
        tell(bdry_out),
        write('bdry('),
        write([N1,N2]),write(','),
        write(B),
        write(').'), nl.

write_to_bdry_file(bdry,B) :-
        write_flag(no_write), !.
write_to_bdry_file(bdry,B) :-
        write_flag(write),
        tell(bdry_out),
        write('bdry('),
        write(B),
        write(').'), nl.


/*
 *  Output graphics instructions in "figure" format to file "bdry_fig"
 */
output_to_figure_file :-
        tell(bdry_fig),
        write_heading(bdry),
        bdry(BdryList),
        write_to_fig_file(bdry,BdryList), !.

write_to_fig_file(title) :-
        tell(bdry_fig),
        title(Text),
        assertz(subtitle('')),
        subtitle(Text2),
        assertz(width_of_title(10)),     /* Default width */
        width_of_title(W),
        Indent is 4.25 - W/16,
        write(drawtext),nl,
        write(Indent),write(' '),write(10.3),write(' '),write(0),nl,
        write(Text),nl,
        write(drawtext),nl,
        write(Indent),write(' '),write(9.9),write(' '),write(0),nl,
        write(Text2),nl, !.
write_to_fig_file(title) :- !.

write_heading(bdry) :-
        tell(bdry_fig),
        write(linestyle),nl, write(1),nl,
        write(linewidth),nl, write(0.01),nl.

write_heading(region) :-
        tell(bdry_fig),
        write(linestyle),nl, write(2),nl,
        write(linewidth),nl, write(0.03),nl.

write_to_fig_file(bdry,[X1,Y1,X2,Y2|Rest]) :-
        tell(bdry_fig),
        draw_line(X1,Y1,X2,Y2),
        write_to_fig_file(bdry,[X2,Y2|Rest]).
write_to_fig_file(bdry,_).
write_to_fig_file(inv_bdry,[X1,Y1,X2,Y2|Rest]) :-
        tell(bdry_fig),
        draw_line_inv(X1,Y1,X2,Y2),
```

```
        write_to_fig_file(inv_bdry,[X2,Y2|Rest]).
write_to_fig_file(inv_bdry,_).

write_to_fig_file(goal,[X,Y]) :-
        tell(bdry_fig),
        scale_coords(X,Y,X1,Y1),
        write(linestyle),nl, write(1),nl,
        write(circle),nl,
        write(X1),write(' '),write(Y1),write(' '),write(0),nl,
        write(0.04),nl, !.
write_to_fig_file(inv_goal,[X,Y]) :-
        tell(bdry_fig),
        scale_coords_inv(X,Y,X1,Y1),
        write(linestyle),nl, write(1),nl,
        write(circle),nl,
        write(X1),write(' '),write(Y1),write(' '),write(0),nl,
        write(0.04),nl, !.

write_to_fig_file(region,[X1,Y1|Rest]) :-
        tell(bdry_fig),
        write_to_fig_file2(region,[X1,Y1|Rest],Xend,Yend),
        draw_line(X1,Y1,Xend,Yend), !.
write_to_fig_file2(region,[X,Y],X,Y).
write_to_fig_file2(region,[X1,Y1,X2,Y2|Rest],Xend,Yend) :-
        tell(bdry_fig),
        draw_line(X1,Y1,X2,Y2),
        write_to_fig_file2(region,[X2,Y2|Rest],Xend,Yend), !.
write_to_fig_file(inv_region,[X1,Y1,X2,Y2|Rest]) :-
        tell(bdry_fig),
        draw_line_inv(X1,Y1,X2,Y2),
        write_to_fig_file(inv_region,[X2,Y2|Rest]), !.
write_to_fig_file(inv_region,_).

scale_coords(X,Y,X1,Y1) :-                          /* Scales and translates */
        maxX(MaxX),maxY(MaxY),minX(MinX),minY(MinY), /* coords to appropriate */
        X1 is 1 + (6.5*(X-MinX)/(MaxX-MinX)),        /* output coord system   */
        Y1 is 1 + (9*(Y-MinY)/(MaxY-MinY)), !.
scale_coords_inv(X,Y,X1,Y1) :-                       /* also reflects the    */
        maxX(MaxX),maxY(MaxY),minX(MinX),minY(MinY), /* coords about the     */
        X1 is 7.5 - (6.5*(X-MinX)/(MaxX-MinX)),      /* vertical line X=4.25 */
        Y1 is 1 + (9*(Y-MinY)/(MaxY-MinY)), !.

draw_line(X1,Y1,X2,Y2) :-
        scale_coords(X1,Y1,X1b,Y1b),
        scale_coords(X2,Y2,X2b,Y2b),
        write(polyline),nl, write(2),nl,
        write(X1b),write(' '),write(Y1b),write(' '),write(0),nl,
        write(X2b),write(' '),write(Y2b),write(' '),write(0),nl, !.
draw_line_inv(X1,Y1,X2,Y2) :-
        scale_coords_inv(X1,Y1,X1b,Y1b),
        scale_coords_inv(X2,Y2,X2b,Y2b),
        write(polyline),nl, write(2),nl,
        write(X1b),write(' '),write(Y1b),write(' '),write(0),nl,
        write(X2b),write(' '),write(Y2b),write(' '),write(0),nl, !.
```

```
/*************************************************************************
 *************************************************************************
 *
 *   "bgmd22"
 *
 *   File "bgmapdata" has terrain and goal data used by "bdrygen".
 *
 *************************************************************************
 *************************************************************************/

/*   "region_vertices" lists the vertices of one HCA
 *   in clockwise order.  First point listed can be any
 *   of the vertices.
 */
region_vertices([4,20,30,70,40,71,60,30,36,8]).

title('Example 22').

goal_point(35,84).


/*   "_cost" is the time required to travel one unit of distance.
 *   Note that this is the inverse of the "cost" used in
 *   the "sls" code.
 */
interior_cost(2).
exterior_cost(1).


/*
 *   "optimal_path" is a temporary set of predicates which specify
 *   the optimal path list from each vertex in the map.
 *   Eventually, it will be replaced by a rule which computes
 *   the optimal path using a path-finding routine such as
 *   "sls" or "rrr".
 */

optimal_path([4,20,c(1),30,70,c(1),35,84]).
optimal_path([30,70,c(1),35,84]).
optimal_path([36,8,c(1),60,30,c(1),35,84]).
optimal_path([60,30,c(1),35,84]).
optimal_path([40,71,c(1),35,84]).
```

REFERENCES

1. Lozano-Perez, T., and Wesley, M. A., "An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles", *Communications of the ACM*, Vol. 22, No. 10, 1979, pp. 560-570.

2. Rowe, N. C., "Roads, Rivers, and Rocks: Optimal Two-Dimensional Route Planning around Linear Features for a Mobile Agent", Technical Report NPS52-87-027, Naval Postgraduate School, Monterey California, 1987 (accepted to *International Journal of Robotics Research*).

3. Mitchell, J. S. B., and Papadimitriou, C. H., "The Weighted Region Problem", Technical Report, Department of Operations Research, Stanford University, Stanford, California, 1986. To appear in *Journal of the ACM*.

4. Mitchell, J. S. B., "A New Algorithm for Shortest Paths Among Obstacles in the Plane", School of Operations Research and Industrial Engineering, Cornell University, Ithaca, New York, 1989, to appear in *Journal of the ACM*.

5. Barr, A., and Feigenbaum, E. A., *The Handbook of Artificial Intelligence*, pp. 19-72, Addison-Wesley, Inc., 1986.

6. Rowe, N. C., *Artificial Intelligence Through Prolog*, Prentice-Hall, 1988.

7. Rich, E., *Artificial Intelligence*, McGraw-Hill, New York, 1983.

8. Preparata, F. P., and Shamos, M. I., *Computational Geometry, An Introduction*, Springer-Verlag, 1988.

9. Wu, P. K., Computer Science Seminar, Naval Postgraduate School, Monterey California, April, 1989.

10. Lee, D. T., and Drysdale, R. L., "Generalization of Voronoi Diagrams in the Plane", *SIAM Journal of Computing*, Vol 10, No. 1, pp. 73-87.

11. Chew, L. P., and Drysdale, R. L., "Voronoi Diagrams Based on Convex Distance Functions", Department of Mathematics and Computer Science, Dartmouth College, Hanover, Hew Hampshire, 1985.

12. Aurenhammer, F., and Edelsbrunner, H., "An Optimal Algorithm for Constructing the Weighted Voronoi Diagram in the Plane", *Pattern Recognition*, Vol. 17, No. 2, pp. 251-257, 1984.

13. Aronov, B., "On the Geodesic Voronoi Diagram of Point Sites in a Simple Polygon", *Proceedings of the Third Annual ACM Symposium on Computational Geometry*, Waterloo, Ontario, pp. 39-49, 1987.

14. Brooks, R. A., "Solving the Find-Path Problem by Good Representation of Free Space", *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 13, No. 3, 1983, pp. 190-197.

15.Mitchell, J. S. B., "An Algorithmic Approach to Some Problems in Terrain Navigation", *Artificial Intelligence*, Vol. 37, 1988, pp. 171-201.

16. Turnage, G. W., and Smith, J. L., "Adaptation and Condensation of the Army Mobility Model for Cross-Country Mobility Mapping", Technical Report GL-83-12, Geotechnical Laboratory, U. S. Army Engineer Waterways Expoeriment Station, Vicksburg, Mississippi, 1983.

17. Pearl, J. *Heuristics, Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, Reading, 1984.

18. Kirkpatrick, S., Gelatt Jr., C.D., and Vecchi, M.P., "Optimization by Simulated Annealing", *Science*, Vol. 220, No. 4598, 13 May 1983, pp 671-680.

19. Mitchell, J. S. B., and Kiersey, D. M., "Planning Strategic Paths Through Variable Terrain Data", *SPIE Vol 485 Applications of Artificial Intelligence*, 1984, pp. 172-179.

20. Richbourg, R. F., "Solving A Class of Spatial Reasoning Problems: Minimal-cost Path Planning in the Cartesian Plane,: Ph.D. Dissertation, Naval Postgraduate School, Monterey, California, 1987.

21. Richbourg, Robert F., "Path-Planning Algorithm Implementations", Technical Report NPS52-87-022, Naval Postgraduate School, Monterey California, 1987.

22. Garey, M. R., and Johnson, D. S., *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York, New York, 1979, pp. 90-92.

23. Kiersey, D., Mitchell, J. S. B., Payton, D., and Preyss, E., "Path Planning for Autonomous Vehicles", *SPIE, Vol. 485, Applications of Artificial Intelligence*, 1984.

24. Kirkpatrick, S., Gelatt Jr., C. D., and Vecchi, M. P., "Optimization by Simulated Annealing", *Science*, Vol. 220, No. 4598, 1983, pp. 671-680.

25. Lindsay, C., "Automatic Planning of Safe and Efficient Robot Paths Using an Octree Representation of a Configured Space",, IEEE International Conference on Robotics and Automation, Raleigh, North Carolina, 1987.

26. Vossepoels, A.M., & Smeulders, S.W.M., "Vector Code Probability and Metrication Error in the Representation of Straight Lines of Finite Length", *Computer Graphics and Image Processing, Vol. 20*, pp.347-364, 1982.

27. Quek, F. K. H., Franklin, R. F., and Pont, F., "A Decision System for Autonomous Robot Navigation Over Rough Terrain", *Proceedings SPIE Applications of Artificial Intelligence,* boston, Massachusetts, 1985.

28. Guibas, L., and Hershberger, J., "Optimal Shortest Path Queries in a Simple Polygon", *Proceedings Third Annual ACM Conference on Computational Geometry*, pp. 50-63, Waterloo, Ontario, 1987.

29. Linden, T. A., Marsh, J. P., and Dove, D. L., "Architecture and Early Experience With Planning for the ALV", *Proceedings, IEEE International Conference on Robotics and Automation*, San Francisco, California, 1985, pp. 2035-2042.

30. Jorgenson, C., "Robot Navigation Using Neural Networks", Workshop on Advanced Computer Architectures for Robotics and Machine Intelligence: Neural Networks and Neurocomputers, IEEE International Conference on Robotics and Automation, Raleigh, North Carolina, 1987.

31. Lee, D. T., "Proximity and Reachability in the Plane", Ph.D. Dissertation, Technical Report ACT-12, Coordinated Science Library, University of Illinois, 1978.

32. Mitchell, J. S. B., "Planning Shortest Paths", Ph.D. Dissertation, Department of Operations Research, Stanford University, 1986.

33. Welzl, E., "Constructing the Visibility Graph for n Line Segments in $O(n^2)$ Time", *Information Processing Letters 20*, Elsevier Science Publishers, B. V., (North-Holland), 1985.

34. Asano, Takao, Asano, Tetsuo, Guibas, L., Hershberger, J. and Imai, H., "Visibility-Polygon Search and Euclidean Shortest Paths", *Proceedings 26th Symposium on Foundations of Computer Science*, pp. 155-164, 1985.

35. Ghosh, S. K., and Mount, D. M., "An Output Sensitive Algorithm for Computing Visibility Graphs", Technical Report CS-TR-1874, Department of Computer Science, University of Maryland, 1987.

36. Aho, A. V., Hopcroft, J. E., and Ullman, J. D., *Data Structures and Algorithms*, Addison-Wesley, Inc., 1987.

37. Gewali, S., Meng, A., Mitchell, J. S. B., and Ntafos, S., "Path Planning in $0/1/\infty$ Weighted Regions With Applications", Extended Abstract in *Proceedings of the Fourth Annual ACM Conference on Computational Geometry*, Urbana-Champaign, Illinois, pp. 266-278, 1988.

38. Richbourg, R. F., Rowe, N. C., Zyda, M. J., and McGhee, R., "Solving Global Two-Dimensional Routing Problems Using Snell's Law and A* Search", *Proceedings IEEE International Conference on Robotics and Automation,* pp. 1631-1636, Raleigh, North Carolina, 1987.

39. Rowe, N. C., and Richbourg, R. F., "An Efficient Snell's-Law Method for Optimal-Path Planning Across Multiple Two-Dimensional Irregular Homogeneous-Cost Regions", Technical Report NPS52-88-017, Naval Postgraduate School, Monterey California, 1988.

40. Ross, R., ..., Ph.D. Dissertation, Naval Postgraduate School, Monterey, California, 1989.

41. Lee, D. T., and Preparata, F. P., "Euclidean Shortest Paths in the Presence of Rectilinear Barriers", *Networks* Vol. 14, pp. 393-410, 1984.

42. Reif, J. H., and Storer, J. A., "Shortest Paths in the Plane with Polyhedral Obstacles", Technical Report CS-85-121, Computer Science Department, Brandeis University, Waltham, Massachusetts, 1985.

43. Payton, D. W., "Internalized Plans: A Representation for Action Resources", presented at the Workshop on Representation and Learning in an Autonomous Agent, Faro, Portugal, 1988.

44. Wade, R., ... , M.S. Thesis, Naval Postgraduate School, Monterey, California, 1989.

45. Pountain, D., "Digital Paper", *Byte*, Vol. 14 No. 2, pp. 274-280, 1989.

46. Hillier, F. S., and Lieberman, G. J., *Introduction to Operations Research*, p. 271, Holden-Day, Inc., 1980.

REFERENCES.

1. Lozano-Perez, T., and Wesley, M. A., "An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles", *Communications of the ACM*, Vol. 22, No. 10, 1979, pp. 560-570.

2. Rowe, N. C., "Roads, Rivers, and Rocks: Optimal Two-Dimensional Route Planning around Linear Features for a Mobile Agent", Technical Report NPS52-87-027, Naval Postgraduate School, Monterey California, 1987 (accepted to *International Journal of Robotics Research*).

3. Mitchell, J. S. B., and Papadimitriou, C. H., "The Weighted Region Problem", Technical Report, Department of Operations Research, Stanford University, Stanford, California, 1986. To appear in *Journal of the ACM*.

4. Mitchell, J. S. B., "A New Algorithm for Shortest Paths Among Obstacles in the Plane", School of Operations Research and Industrial Engineering, Cornell University, Ithaca, New York, 1989, to appear in *Journal of the ACM*.

5. Barr, A., and Feigenbaum, E. A., *The Handbook of Artificial Intelligence*, pp. 19-72, Addison-Wesley, Inc., 1986.

6. Rowe, N. C., *Artificial Intelligence Through Prolog*, Prentice-Hall, 1988.

7. Rich, E., *Artificial Intelligence*, McGraw-Hill, New York, 1983.

8. Preparata, F. P., and Shamos, M. I., *Computational Geometry, An Introduction*, Springer-Verlag, 1988.

9. Wu, P. K., Computer Science Seminar, Naval Postgraduate School, Monterey California, April, 1989.

10. Lee, D. T., and Drysdale, R. L., "Generalization of Voronoi Diagrams in the Plane", *SIAM Journal of Computing*, Vol 10, No. 1, pp. 73-87.

11. Chew, L. P., and Drysdale, R. L., "Voronoi Diagrams Based on Convex Distance Functions", Department of Mathematics and Computer Science, Dartmouth College, Hanover, Hew Hampshire, 1985.

12. Aurenhammer, F., and Edelsbrunner, H., "An Optimal Algorithm for Constructing the Weighted Voronoi Diagram in the Plane", *Pattern Recognition*, Vol. 17, No. 2, pp. 251-257, 1984.

13. Aronov, B., "On the Geodesic Voronoi Diagram of Point Sites in a Simple Polygon", *Proceedings of the Third Annual ACM Symposium on Computational Geometry*, Waterloo, Ontario, pp. 39-49, 1987.

14. Brooks, R. A., "Solving the Find-Path Problem by Good Representation of Free Space", *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 13, No. 3, 1983, pp. 190-197.

15. Mitchell, J. S. B., "An Algorithmic Approach to Some Problems in Terrain Navigation", *Artificial Intelligence*, Vol. 37, 1988, pp. 171-201.

16. Turnage, G. W., and Smith, J. L., "Adaptation and Condensation of the Army Mobility Model for Cross-Country Mobility Mapping", Technical Report GL-83-12, Geotechnical Laboratory, U. S. Army Engineer Waterways Expoeriment Station, Vicksburg, Mississippi, 1983.

17. Pearl, J. *Heuristics, Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, Reading, 1984.

18. Kirkpatrick, S., Gelatt Jr., C.D., and Vecchi, M.P., "Optimization by Simulated Annealing", *Science*, Vol. 220, No. 4598, 13 May 1983, pp. 671-680.

19. Mitchell, J. S. B., and Kiersey, D. M., "Planning Strategic Paths Through Variable Terrain Data", *SPIE Vol 485 Applications of Artificial Intelligence*, 1984, pp. 172-179.

20. Richbourg, R. F., "Solving A Class of Spatial Reasoning Problems: Minimal-cost Path Planning in the Cartesian Plane,: Ph.D. Dissertation, Naval Postgraduate School, Monterey, Califomia, 1987.

21. Richbourg, Robert F., "Path-Planning Algorithm Implementations", Technical Report NPS52-87-022, Naval Postgraduate School, Monterey Califomia, 1987.

22. Garey, M. R., and Johnson, D. S., *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York, New York, 1979, pp. 90-92.

23. Kiersey, D., Mitchell, J. S. B., Payton, D., and Preyss, E., "Path Planning for Autonomous Vehicles", *SPIE, Vol. 485, Applications of Artificial Intelligence*, 1984.

24. Kirkpatrick, S., Gelatt Jr., C. D., and Vecchi, M. P., "Optimization by Simulated Annealing", *Science*, Vol. 220, No. 4598, 1983, pp. 671-680.

25. Lindsay, C., "Automatic Planning of Safe and Efficient Robot Paths Using an Octree Representation of a Configured Space",, IEEE International Conference on Robotics and Automation, Raleigh, North Carolina, 1987.

26. Vossepoels, A.M., & Smeulders, S.W.M., "Vector Code Probability and Metrication Error in the Representation of Straight Lines of Finite Length", *Computer Graphics and Image Processing, Vol. 20*, pp. 347-364, 1982.

27. Quek, F. K. H., Franklin, R. F., and Pont, F., "A Decision System for Autonomous Robot Navigation Over Rough Terrain", *Proceedings SPIE Applications of Artificial Intelligence,* boston, Massachusetts, 1985.

28. Guibas, L., and Hershberger, J., "Optimal Shortest Path Queries in a Simple Polygon", *Proceedings Third Annual ACM Conference on Computational Geometry*, pp. 50-63, Waterloo, Ontario, 1987.

29. Linden, T. A., Marsh, J. P., and Dove, D. L., "Architecture and Early Experience With Planning for the ALV", *Proceedings, IEEE International Conference on Robotics and Automation*, San Francisco, California, 1985, pp. 2035-2042.

30. Jorgenson, C., "Robot Navigation Using Neural Networks", Workshop on Advanced Computer Architectures for Robotics and Machine Intelligence: Neural Networks and Neurocomputers, IEEE International Conference on Robotics and Automation, Raleigh, North Carolina, 1987.

31. Lee, D. T., "Proximity and Reachability in the Plane", Ph.D. Dissertation, Technical Report ACT-12, Coordinated Science Library, University of Illinois, 1978.

32. Mitchell, J. S. B., "Planning Shortest Paths", Ph.D. Dissertation, Department of Operations Research, Stanford University, 1986.

33. Welzl, E., "Constructing the Visibility Graph for n Line Segments in $O(n^2)$ Time", *Information Processing Letters 20*, Elsevier Science Publishers, B. V., (North-Holland), 1985.

34. Asano, Takao, Asano, Tetsuo, Guibas, L., Hershberger, J. and Imai, H., "Visibility-Polygon Search and Euclidean Shortest Paths", *Proceedings 26th Symposium on Foundations of Computer Science*, pp. 155-164, 1985.

35. Ghosh, S. K., and Mount, D. M., "An Output Sensitive Algorithm for Computing Visibility Graphs", Technical Report CS-TR-1874, Department of Computer Science, University of Maryland, 1987.

36. Aho, A. V., Hopcroft, J. E., and Ullman, J. D., *Data Structures and Algorithms*, Addison-Wesley, Inc., 1987.

37. Gewali, S., Meng, A., Mitchell, J. S. B., and Ntafos, S., "Path Planning in $0/1/\infty$ Weighted Regions With Applications", Extended Abstract in *Proceedings of the Fourth Annual ACM Conference on Computational Geometry*, Urbana-Champaign, Illinois, pp. 266-278, 1988.

38. Richbourg, R. F., Rowe, N. C., Zyda, M. J., and McGhee, R., "Solving Global Two-Dimensional Routing Problems Using Snell's Law and A* Search", *Proceedings IEEE International Conference on Robotics and Automation*, pp. 1631-1636, Raleigh, North Carolina, 1987.

39. Rowe, N. C., and Richbourg, R. F., "An Efficient Snell's-Law Method for Optimal-Path Planning Across Multiple Two-Dimensional Irregular Homogeneous-Cost Regions", Technical Report NPS52-88-017, Naval Postgraduate School, Monterey California, 1988 (accepted to *International Journal of Robotics Research*).

40. Ross, R., "Planning Minimum-Energy Paths in an Off-Road Environment with Anisotropic Traversal Costs and Motion Constraints", Ph.D. Dissertation, Naval Postgraduate School, Monterey, California, 1989.

41. Lee, D. T., and Preparata, F. P., "Euclidean Shortest Paths in the Presence of Rectilinear Barriers", *Networks* Vol. 14, pp. 393-410, 1984.

42. Reif, J. H., and Storer, J. A., "Shortest Paths in the Plane with Polyhedral Obstacles", Technical Report CS-85-121, Computer Science Department, Brandeis University, Waltham, Massachusetts, 1985.

43. Payton, D. W., "Internalized Plans: A Representation for Action Resources", presented at the Workshop on Representation and Learning in an Autonomous Agent, Faro, Portugal, 1988.

44. Wade, R., M.S. Thesis, Naval Postgraduate School, Monterey, California, 1989.

45. Pountain, D., "Digital Paper", *Byte*, Vol. 14 No. 2, pp. 274-280, 1989.

46. Hillier, F. S., and Lieberman, G. J., *Introduction to Operations Research*, p. 271, Holden-Day, Inc., 1980.

DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Library, Code 0142 Naval Postgraduate School Monterey, CA 93943	2
Dr. Neil C. Rowe, Code 52Rp Department of Computer Science Naval Postgraduate School Monterey, CA 93943	10
Dr. Maurice D. Weir, Code 53Wc Department of Mathematics Naval Postgraduate School Monterey, CA 93943	2
Dr. C. Thomas Wu, Code 52Wq Department of Computer Science Naval Postgraduate School Monterey, CA 93943	2
Dr. Yuh-Jeng Lee, Code 52Le Department of Computer Science Naval Postgraduate School Monterey, CA 93943	2
Dr. Donald R. Barr Valuation Technology, Inc. 6800 Garden Road Monterey, CA 93940	2
Dr. Robert B. McGhee, Code 52Mz Department of Computer Science Naval Postgraduate School Monterey, CA 93940	2

Dr. Man-Tak Shing, Code 52Sh
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943 2

Major Robert S. Alexander
2904 So. 14th Street
Leavenworth, KS 66044 15

614-583

Thesis
A3711 Alexander
c.1 Construction of optimal-path maps for homogeneous-cost-region path-planning problems.

thesA3711
Construction of optimal-path maps for ho
3 2768 000 86362 5
DUDLEY KNOX LIBRARY